TDS-91-04

Cost-effectiveness of Passing Lanes: Programmer's Manual

Ontario

Ministry of Transportation

Research and Development Branch

Ontario
Ministry of Transportation
Research and Development Branch

Documentation Page

TDS-91-04

Cost-effectiveness of Passing Lanes: Programmer's Manual

Author(s): A.M. Khan, N.M. Holtz, Carleton University

Date Published: September, 1991

Published by: Research and Development Branch, MTO

Contact Person: Alex Ugge, Pavements & Roadway Office
(416) 235- 4698

Abstract: Program PCL is an interactive, screen-oriented, menu-driven, PC-based software package for the anaylsis of the effect of adding passing or climbing lanes to selected locations along two-lane rural highways. The main analysis functions are: a) choosing the start and end locations of a climbing lane based on the speed differential of a heavily loaded truck; b) predicting the effects on level of service and average speed of adding a passing or climbing lane; c) performing a cost effectiveness analysis based on reduction of delay and reduction in accidents. PCL provides extensive on-line help for the novice or inexperienced user, and is menu-driven for ease of use.

The Programmer's Manual provides a general overview of program logic, and extensive documentation of the internal workings of some portions of the PCL program. It is intended for use by programmers who wish to make modifications to the program.

The PCL program is written in ANSI Standard C, and comprises about 50 source files totalling approximately 17,000 lines and over 450K bytes (exclusive of screen handling libraries). The extensive use of PC-specific screen manipulation means that porting to other (non-PC) environments will be difficult.

Comments: Programmer's Manual for PCL Computer Program. Final report for research project (TDS-91-02) and User's Manual (TDS-91-03) also available.

Key Words: passing lane, climbing lane, grade, two-lane road, speed-distance, cost-effectiveness, computer program, programmer documentation.

Cost-effectiveness of Passing Lanes: Programmer's Manual

A.M. Khan
Principal Investigator

N.M. Holtz
Investigator

Carleton University

Published by
The Research and Development Branch
Ontario Ministry of Transportation

For additional copies, contact:
The Editor, Technical Publications
Room 320, Central Building
1201 Wilson Avenue
Downsview, Ontario
Canada M3M 1J8

Telephone: (416) 235-3480
Fax: (416) 235-4872

September 1991

Table of Contents

Table of Contents (continued)

Preface

This document is the Programmer's Manual for PCL. It is intended for use by knowledgeable programmers to aid them in making modifications to and performing normal maintenance of the program. This manual provides only an overview of general logic and some specific documentation of a few of the utility modules. Many of the details can only be learned by reading the source code.

The User's Manual provides much in the way of general information, and the programmer is referred to that document, as well.

Program PCL[1] is an interactive, screen-oriented PC-based software package for the analysis of the effect of adding passing or climbing lanes to selected locations along two-lane rural highways. The main analysis functions are:

- choosing the start and end locations of the climbing lane based on the speed differential of a heavily loaded truck.
- the effects on level of service and average speed when a passing or climbing lane is added.
- cost effectiveness analysis based on reduction of delay and reduction in accidents.

PCL was developed by the authors at the Department of Civil Engineering, Carleton University, Ottawa, Ontario under contract for Projects 25193, "*Cost-Effectiveness of Climbing Lanes: Safety, Level of Service and Cost Factors*", and 21215, "*Priority Analysis of Passing/Climbing Lane Solutions on Two-Lane Rural Roads*", both for the Ministry of Transportation, Province of Ontario.

The PCL programs and related documentation and data files are the property of the Ministry of Transportation, Province of Ontario.

[1] PCL *could* be pronounced "pickle".

Part I

General Overview

1 Introduction

1.1 Language, Libraries and Portability

PCL is written in the C language, specifically Turbo C 2.0[1] by Borland.[2] It makes use of an extensive screen–handling library called Turbo C TOOLS 2.0 from Blaise Computing.[3]

The resulting program is written for and runs only on the "IBM PC" class of machines (PC and AT-compatibles).

While a serious attempt has been made to code using only ANSI standard C, this has not been fully tested. By its nature, because of the extensive, non-portable screen handling library, PCL will not port easily to other, non-PC, environments.

1.2 Licensed Software

The normal software license from Borland allows distribution of binaries created with the TURBO-C compiler, and also allows distribution of the binary Borland Graphics Interface (BGI) graphics device driver files and the binary character font (CHR) files.

The software license from Blaise Computing allows distribution of programs containing linked routines from the library, but does not allow distribution of the libraries themselves.

Therefore, the distribution of PCL does *not* contain a C compiler, nor any of the Blaise libraries.

See section 1.6, below, for a list of software required when making changes to PCL.

---

[1] TURBO C is a trademark of Borland International, Inc.

[2] Borland International, 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95066-0001.

[3] Blaise Computing Inc., 2560 Ninth Street, Suite 316, Berkeley, CA 94710, (415) 540-5441.

1.3 Other Software Tools

The author made use of many other tools in the development of PCL. Their use is not essential, but it is very convenient. The various tools used were:

DIFF – a standard Unix utility that prepares a list of the differences between files.

PDMAKE – a public–domain "make" utility that is quite compatible with early UNIX[4] 'make'. Used to control as-needed compilation, linking and re-building. The TURBO C make is not used because it is slightly less compatible with Unix make, but more importantly, because it does not allow command line i/o redirection[5] and we depend on that to re-create linker response files.

TEX and **LATEX** – specifically PCTEX. A document typesetting language. Used to format the documentation and help files.

TEXINDEX – an index sorting and merging utility designed to work with TEX indices. Free software from the GNU project.[6]

DVI2PS – translates TEX DVI files to Postscript[7] for printing on a Postscript printer. This is *not* the public domain version of DVI2PS that is widely available, but is one developed by one of the authors for Digital Composition Systems, Inc., Phoenix, AZ. The major repercussion is in the support for including Postscript graphics (the many screen displays for the figures) through TEX's `\special` command; other DVI translators will do it differently or not at all. Only of concern when you re-typeset the manuals, and then the worst that should happen is that the figures will be missing (but proper blank space will be left).

Vanilla SNOBOL 4 – an implementation of the SNOBOL programming language for PC's, it is freeware distributed by Catspaw.[8] Used extensively for text manipulation – specifically translation from TEX source to Blaise help file format, and processing of indices for the manuals.

[4] UNIX is a trademark of AT&T.
[5] What a botch!
[6] Free Software Foundation, ???.
[7] Postscript is a trademark of Adobe, Inc.
[8] Catspaw, Inc., P.O. Box 1123, Salida, Colorado, 81201. (719)539-3884.

PKZIP, PKUNZIP – file compression, archiving and de-archiving software. Used to reduce the number of diskettes required to hold a PCL distribution. This is shareware from PKWARE.[9]

SD2PS – converts PCL screen dump files (with attributes) to Postscript. Written by the author of PCL. Used to generate most of the figures for the manuals.

XFUNNAME, XDOC – are two small C programs that are useful for extracting the documentation at the front of each function in the source files, and building a TEX input source file, sorted by function name.

DIFF, PDMAKE, TEXINDEX, SNOBOL 4, PKZIP/PKUNZIP, SD2PS XFUNNAME and XDOC are included with the source distribution of PCL; the various licensing agreements and copyrights allow this. In addition, the source (in C) for PDMAKE, TEXINDEX and SD2PS is also included.

PCTEX may be obtained separately from Personal TEX Inc.,[10] although any version of TEX and LATEX (version 2.09 or later) on any machine should work.

A completely compatible DVI2PS is probably not available from any source.

1.4 Directory Structure

The structure and contents of the PCL directory tree is as follows:

```
--pcl---|
        |--doc------|
        |           |--eg
        |--install
        |--sd2ps
        |--src
        |--tables
        |--tools----|
                    |--src
```

doc Contains the TEX source for the documentation – the printed manuals as well as the on-line help database.

[9] PKWARE, Inc., 7032 Ardara Avenue, Glendale, WI 53209.

[10] Personal TEX Inc., 12 Madrona Ave., Mill Valley, CA, 94941. (415)388-8853.

`doc/eg` Contains example site files, analysis print files and screen dumps for the figures for the manuals.

`install` Contains the `install` program (including source), as well as command files to construct binary and source distribution floppies.

`sd2ps` Contains the source for program `SD2PS` – to convert screen dump files to Postscript.

`src` Contains all the C source files for the executable modules. Does *not* include any of the Blaise library routines, except for two that have had bugs corrected.

`tables` Contains the speed-distance tables used to compute acceleration and deceleration, and thus speed versus distance, of trucks on grade.

`tools` Contains miscellaneous tools that can be distributed with the PCL source. Includes `pdmake`, `snobol4`, etc.

`tools/src` Contains the source for those tools for which source is available.

1.5 PCL Structure

PCL consists of a number of modules that the user must execute in turn. For the most part, these are implemented as separate executable programs, all invoked by the main controlling program `PCL`. The separate executable modules are:

1. `pcl` – main executable.
2. `defsite` – For input of roadway and traffic characteristics.
3. `defcl` – For computation of climbing lane locations.
4. `estperf` – For performance estimate of climbing lane.
5. `estpass` – For performance estimate of passing lane.
6. `effmeas` – For effectiveness measures computation.

In contrast, the "Site/File Operations" module that is shown on the main `pcl` menu is actually part of the `pcl` program.

1.6 Making Changes

To make any modifications to PCL program logic, you will need the following:

- a licensed version of "TURBO C", version 2.0 or later, from Borland. TURBO C++ 1.0 will also work (but note that PCL is coded in *C*, not *C++*).
- a licensed version of "Turbo C TOOLS/2.0" from Blaise.

The following are highly recommended, but not essential:

- a UNIX-compatible 'make' facility (such as **pdmake**, which is distributed with PCL). The version of **make** included with TURBO-C is *not* compatible.

The remainder of this section outlines what you must do after you make a change to any of the component source files. In all further discussion, directory "**\pcl**" is assumed to be the root PCL directory. If your root directory is different, substitute its full pathname for **\pcl**.

There are several binaries that have to be rebuilt when modifications are made:

1. **\pcl\pcl.exe** – main executable.
2. **\pcl\defsite.exe** – For input of roadway and traffic characteristics.
3. **\pcl\defcl.exe** – For computation of climbing lane locations.
4. **\pcl\estperf.exe** – For performance estimate of climbing lane.
5. **\pcl\estpass.exe** – For performance estimate of passing lane.
6. **\pcl\effmeas.exe** – For effectiveness measures computation.
7. **\pcl\pclhlp.hlp** – The on-line help database.
8. **\pcl\doc\manual.dvi** – The typeset user's manual.
9. **\pcl\doc\progman.dvi** – The typeset programmer's manual.

With the exception of the manuals, the **makefile** will cause everything to be rebuilt automatically when any of the source is edited.

1.6.1 Fixing Blaise Tools Bugs

There are two routines in the Blaise Tools/2.0 library that have to be fixed before PCL can be rebuilt. They are in Blaise source files `edbase.c` and `edremkey.c`. The Blaise licensing agreements prohibit the distribution of the repaired source code for those two routines. File "`src\patches.bla`" gives the differences of the fixed versions from the original versions. You will have to change your versions to match.

The `patches.bla` file was prepared by "diff" (-c option). Briefly, diff compares an original (*file1*) and a changed version (*file2*). Each changed area is identified, and the lines from *file1* are output, then the corresponding lines from *file2*. Lines deleted from *file1* are marked with a "-", lines added to *file2* are marked with a "+", and lines changed are marked with a "!" in both.

To change the Blaise source files:

1. copy the source files `edbase.c` and `edremkey.c` to the PCL source directory (normally `\pcl\src`).

2. apply the patches in file `\pcl\src\patches.bla` to the two files.

1.6.2 Re-Compiling PCL and Modules

If you use "pdmake", it is simple. To recompile and re-link those modules that have been changed (and also to rebuild the help database, if necessary), simply type:

```
cd \pcl\src
pdmake
```

The `pdmake` command verb can be followed by one of `pcl`, `defsite`, `defcl`, `estperf`, `effmeas` or `helpfile` if you wish to rebuild only one of the components. There is usually no advantage to this, though.

If, for some reason, you choose not to use "pdmake", then you have to compile files with a single command line like:

```
tcc -c -N -v -ml -f -O -G -wnod -wrvl -wpro -wsig
                          -wcln -wucp -w-par pcl.c
```

Note that we use the large memory model.

To re-link a specific module, such as `pcl.exe`:

```
tlink /x /c /v \turbo\tc\c0l.obj @pcll.lnk
```

For each "*prog*`.exe`" file mentioned above, there is a "*prog*`1.lnk`" linker response file to build it. Note that the linker response files, such as `pcll.lnk`, are rebuilt automatically by `pdmake` whenever the `makefile` is modified.

1.6.3 Re-Printing Documentation

Files `manual.tex` and `progman.tex` in the "`\pcl\doc`" directory are the master files for the User's Manual and Programmer's Manual. They "`\include{}`" all of the other necessary LaTeX source files.

You can edit both `manual.tex` and `progman.tex` to use the LaTeX `\includeonly{}` commands to format only parts of the document. Note that if you rebuild the index in these cases, you will get an index only for the parts you printed.

For the User's Manual, do:

```
cd \pcl\doc
latex manual
manindx
latex manual
```

or:

```
cd \pcl\doc
pdmake userman
```

For the Programmer's Manual, do:

```
cd \pcl\doc
progmsum
progfdoc
latex progman
progindx
latex progman
```

or:

```
cd \pcl\doc
pdmake progman
```

The `manindx` and `progindx` lines (both commands are small "BAT" files) rebuild the index for each document, and their usage is optional. Also, the second execution of "`latex ...`" is optional if you do not re-build the index and if cross-references do not change. Finally, commands `progmsum`

and **progfdoc** recreate the module summary and function documentation, respectively, from the *C* source files, and need not be given if the source files have not changed significantly.

To print the resulting "**.dvi**" files, see your local TEX documentation. Note that you may have trouble printing the figures along with the document, as they are distributed in Postscript form, and your DVI-to-device translator may not support that.

1.6.4 Re-Building the HELP Database

The LATEX source file "**pclhlp.tex**" contains all of the source for the on-line help database. Extensive comments at the beginning of that file document its rather special structure.

To re-build the help database, the LATEX version must first be translated into a form suitable for input to the Blaise "**BUILDHLP**" program; this is done by a non-trivial SNOBOL program. Then **BUILDHLP** must be run to create the binary version. A makefile exists so that all you have to do is:

```
cd \pcl\doc
pdmake
```

Alternatively, without using **pdmake**, do:

```
cd \pcl\doc
snobol4 tex2hlp /i=pclhlp.tex /o=pclhlp.txt
buildhlp pclhlp.txt
copy pclhlp.hlp ..
del pclhlp.hlp
```

1.6.5 Making Distribution Floppies

To make a set of binary only (no source) distribution floppies:

```
cd \pcl\install
makeflpy
```

This requires you to have prepared four formatted 360 KB (double density) diskettes, or one formatted 1.2 MB (high density) diskette. During the procedure, you will be asked to swap floppies; ignore the message if you are using one high-density diskette.

The **makeflpy** procedure also requires a scratch directory for use when constructing the floppies. The scratch directory requires about 300 KB of space. It is currently fixed as directory "**f:\tmp**" in **makeflpy.bat**; you will

have to edit that file and change all occurrences of `f:\tmp` to something else, or use the DOS `subst` command, as for example:

```
mkdir c:\tmp
subst f: c:\
```

To make the diskettes containing all the distributable source and tool files, do:

```
cd \pcl\install
makesrc a:
```

where `a:` is the drive on which to copy the source. This requires two formatted 1.2 MB (high density) floppies, and they must be formatted ahead of time.

2 Program Source Files

This chapter provides a brief overview of the various *C* source files and the dependencies among them. The following corresponds to program version 2.03.

2.1 The Files (Modules)

`cl_comp.c` *compute climbing lane using speed-distance curves.*

This module uses the speed distance curves provided in data files to compute the speed of a truck along the roadway, and chooses the zones of climbing lane to be those sections where the computed truck speed is lower than specified values (the start point of a zone is the point at which the computed speed drops to the specified value, and the end point of each zone is the point at which the computed speed rises to the specified value). For the purposes of this module, those start and end speeds may be different. In hilly terrain (i.e., up and down grades), this module may compute several zones of climbing lane; the caller must combine these into one zone if it so wishes.

`cl_draw.c` *draw climbing lane.*

This module displays the grade profile, plots truck speed versus distance along the grade, and shows the climbing lane locations chosen. All the graphics are displayed on the CRT screen; there is no provision for hardcopy.

`colours.c` *define colours for all types of screen objects.*

This module contains tables of the colours to use for the various types of text on the different display types (monochrome, CGA, etc). Provided are functions to initialize the tables, and to return the colours to use for each text type. This is the only module where colours are directly referred to, and is thus the only file that need be changed to change colours.

`crt.c` *save and restore state of crt (screen).*

This module provides functions to save and restore the state of the display. It is generally used to restore the screen after a program terminates.

`defcl.c` *defining length and location of climbing lane.*

This main module controls the computation and/or user specification of the location of the start and end of the climbing lane. It requires that module "defsite" has already been run to define the site characteristics. This modules allows the climbing lane locations to be computed on the basis of calculated truck speed drop, and allows the user to specify her own locations. This is the user-interface portion – other modules provide the computation and graphics portions.

`defsite.c` *Define site characteristics.*

This main module is the module that defines site characteristics. A form is presented for filling in by the user; that form can then be saved in a "site file" for use by other modules.

`effmeas.c` *effectiveness measures.*

This main module computes the various effectiveness measures and displays, amoung other things, a cost-benefit analysis. These effectiveness measures can also be plotted in the form of graphs.

`err.c` *error routines.*

The routines in this module provide a simple and consistent set of error handlers. They print messages, and, depending on severity, may terminate program execution. If the status line handler is initialized, the error message is put there, otherwise it is just printed on the error output. When the status line is used to display the message, it is possible for the user to ask for a help window explaining the error.

`estpass.c` *estimate performance of passing lane alternative.*

This main module estimates the performance (delay and accident involvement reduction) of the site with passing lane added. Results are displayed and saved in the site file for use by the effectiveness measures module.

`estperf.c` *estimate performance.*

This main module estimates the performance (delay and accident involvement reduction) of the site with climbing lane added. Results

are displayed and saved in the site file for use by the effectiveness measures module.

`formmeth.c` *form field methods.*

This module contains all the methods for the different types of form fields, plus their support routines. The methods are the seven or so functions that have to be implemented for each field type. The method table for each field type is essentially a vector of pointers to these functions. For the most part, therefore, the routines in this module deal almost exclusively with individual fields.

`forms.c` *forms package.*

This module allows the specification, display and editting of forms. A form is a screen of text containing titles and different types of fields to be filled in. The specification of a form consists of specifying, for each field, its relative location, size, type of field, etc. Macros are provided to ease some of the burden of specification, but it is still not trivial.

A form is specified by an array of structures. Functions are provided to display that form and let a user move around, filling in fields as she goes.

`formsio.c` *forms input/output.*

This module provides functions to write a form's data to a file, and to read it back in again. Form files are readable text files. One field is written on each line; the line is prefixed by the fields "file ID" (see forms package), followed by the subscript range (for vectors) followed by each of the values separated by spaces. The file may contain blank lines and comments.

The form must be described by an array of field descriptors – in fact the identical array that is used for displaying and editing the form.

`f_util.c` *file utility functions.*

This module provides menu and screen-oriented procedures for allowing a user to specify a file name, either by typing the file name or by searching for an existing file.

Also provided are a few file name utilities, and a function to load configuration files.

`graph.c` *low level graphics interface.*

This module contains a number of simple, general purpose graphics routines. It is provided so that porting to other compilers and/or platforms will be easier – only the routines in this module should need changing.

Among the features: 2-dimensional; coordinates are floating point values, in the users coordinate system; user-specifed viewports (portions of the graphics screen) and windows (coordinate range of the users, world, coordinate system).

`help.c` *help utilities.*

This module provides the routines necessary for the on-line help displays. Provided are functions to set, save and restore the help system state. Internal routines provide the logic for the various types of help displays.

This module also provides most of the functionality required for the F10 "options menu", as it already detects "hot keys". The options menu may result in application-dependant functions being called to terminate program, start logging, etc; therefore the application has to specify those functions to this package. In fact, the application can define a number of "hot keys", if it provides a function to handle the key.

`helpsys.c` *help file browser.*

This module provides a simple, menu-oriented browser that allows a user to read some general on-line help. It is usually invoked by the user pressing "Alt-H", but may also be invoked automatically when PCL starts up (at least, for non-expert users).

`init.c` *initialization routines.*

This module provides some general initialization routines – such as routines to change some of the Blaise default key assignments to something more useful.

`m_util.c` *menu utilities.*

This module provides a higher-level menu interface than the standard Blaise tools. A calling program can define a menu using some data initialization statements, and can display and read a selection with one routine call.

A history list of the last few menus is maintained, so that the default selection can be made the same as the seclection made when the menu was last displayed.

`pcl.c` *main driving program.*

This is the main driving module for the PCL system, and is intended to be the primary user interface. This module allows a user to choose a particular site, and to invoke the other modules as desired. The other modules, such as "defsite" (to define the site parameters) and "defcl" (to define or compute the climbing lane location) are separate stand-alone programs. That is, they may be run stand-alone, if desired, although the usual case is to let this module invoke them as the result of a menu selection by the user.

In order to have the other modules communicate a small amount of data back to this module, a "com" (communications) file is written by those modules when they exit, and read by PCL when control returns. That com file contains the site ID, site file name, and description.

As with all modules, certain parameters are read from the configuration file (in this case, PCLPath and Help file name).

`scandir.c` *read disk directories.*

This module provides routines for reading DOS file directories, and determining file names and directory names from them.

`statline.c` *manipulate status line.*

This module provides functions for displaying the status line at the bottom of the screen. Applications can save the current line, display something new, then restore the line.

`tables.c` *provide tables for performance estimates.*

This module provides tables and accessing routines for the various parameters required to compute average speeds on approach and on upgrade sections. This module essentially implements the tables of v/c ratios for approach and for upgrade sections. A number of support routines are provided.

`util.c` *screen/window independant utilitites.*

This module provides some generally useful utilities that are not involved with screen displays. They are mostly conversions (string to numbers) and memory allocation.

`w_util.c` *window utilitites.*

This module provides a simpler, higher-level interface, to the Blaise window management routines. It allows saving and restoring of the current active window, creation of prompters (prompting for textual

or numeric values), confirmers (questions that must be answered with 'yes' or 'no'), and windows that can be written into by the application.

2.2 The File Dependencies

The following are the source file dependencies. That is, each of the main PCL modules is made up of the following source files.

pcl `pcl.c edbase.c edremkey.c colours.c crt.c err.c formmeth.c forms.c formsio.c f_util.c helpsys.c help.c init.c m_util.c scandir.c statline.c util.c w_util.c`

defcl `defcl.c edbase.c edremkey.c cl_comp.c cl_draw.c graph.c colours.c crt.c err.c formmeth.c forms.c formsio.c f_util.c helpsys.c help.c init.c m_util.c scandir.c statline.c util.c w_util.c`

defsite `defsite.c edbase.c edremkey.c colours.c crt.c err.c formmeth.c forms.c formsio.c f_util.c helpsys.c help.c init.c m_util.c scandir.c statline.c util.c w_util.c`

effmeas `effmeas.c graph.c edbase.c edremkey.c colours.c crt.c err.c formmeth.c forms.c formsio.c f_util.c helpsys.c help.c init.c m_util.c scandir.c statline.c util.c w_util.c`

estperf `estperf.c tables.c edbase.c edremkey.c colours.c crt.c err.c formmeth.c forms.c formsio.c f_util.c helpsys.c help.c init.c m_util.c scandir.c statline.c util.c w_util.c`

estpass `estpass.c edbase.c edremkey.c colours.c crt.c err.c formmeth.c forms.c formsio.c f_util.c helpsys.c help.c init.c m_util.c scandir.c statline.c util.c w_util.c`

Part II

Utility Module Details

3 Context Sensitive Help

This chapter describes the program logic used to provide the context-sensitive on-line help messages.

In response to a keystroke (usually either "F1" or "Ctrl-F1" or "Alt-F1"), the context-sensitive help system displays a help window that depends on the current input request (menu selection, form field, or prompt question). Also, "Alt-H" at (almost any) time will invoke the "Introductory Help" system; this allows the user to browse around a fairly extensive set of on-line documentation.

In addition, when entering a numerical value in a form, "Alt-L", will cause the limits to the variable to be displayed in the status line, and will allow "F1" to display a more detailed help message from the help database.

A single displayed help message is referred to as a "topic". The message it self may be shorter or longer than the help window on the screen. If it is longer, the user may scroll it to see other portions.

All help displays are simply short sections of text loaded from a single help file and displayed in a window on the screen.

3.1 Blaise Help Files

The Blaise help file is simply a file of text windows that can be randomly accessed using a 12-character key for each window (topic). The program BUILDHLP, supplied with Turbo C TOOLS from Blaise, creates the binary help file from a text input file.

PCL uses only the Blaise routine `hldisp()` to display help windows; that routine takes as arguments the name of the help file and the 12 character topic ID, displays the topic in a window and allows the user to scroll through it, then deletes the window when the user presses "Esc" or "Enter".

PCL adds a shallow hierarchy to the flat Blaise index structure. It consistently creates a help file ID by concatenating a short module ID with a topic ID (separated by a single space) and uses the result as an index to the help file.

There is a help file module for each program module. In addition, there are other help file modules corresponding to other related sets of help windows. The modules currently defined are:

ID	Module
`MO`	PCL – Site/Analysis Selection
`MOF`	Site/File Operations
`MA`	Roadway and Traffic Characteristics
`MB`	Define Climbing Lane
`MC`	Estimate Performance (Climbing Lane)
`MP`	Estimate Performance (Passing Lane)
`MD`	Effectiveness Measures
`Hlp`	Introductory Help System
`KEYS`	Special Keys
`ERR`	Error Messages
`Fatal`	Fatal Error Messages
`OMenu`	Options (F10) menu
`Lim`	Data Limits Messages

There are many topics within each module. For example, the `KEYS` module contains help windows that describe the various special key functions in different contexts (such as when selecting from a menu, typing a number in a form field, etc).

3.2 Help File Source

File `\pcl\doc\pclhlp.tex` contains the source for the help file. This allows the text to be typeset as part of the User's manual, as well as to be used to build the help database.

LaTeX commands of the form "`\module{`*mod*`}{`*title*`}`" are the headings for each module, and automatically define a topic `Main` for module of ID *mod*.

LaTeX commands of the form "`\topic{`*mod*`}{`*top*`}{`*title*`}`" are the headings for each topic. They define the start of one single topic of ID *top* for module of ID *mod*.

For each topic, only text enclosed in

```
\begin{help}
...
\end{help}
```

becomes part of the help window in the database. This allows the printed documentation to contain additional material that does not become part of the on-line help database.

3.3 General Logic

All input to the program uses the various Blaise TOOLS routines and is accomplished, generally speaking, by displaying an object, such as a menu, on the screen and calling an object-specific procedure to do a "read". For example, `mndsplay()` displays a menu, `mnread()` reads a selection from it, `wnfield()` allows a user to edit a text field and returns the result, etc.

The Blaise TOOLS input routines optionally call an application-supplied "key control function" whenever the keyboard is polled for an input keystroke. The key control function is called *before* the Blaise TOOLS process the keystroke. The key control function can process the key itself and effectively "delete" it from the input, or it can ignore it and let the Blaise TOOLS process it.

It is thus a simple matter of writing a key control function that handles the special function keys such as "`F1`", by displaying the proper help window, then deleting the keystroke.

We actually have two key control functions: one that handles keystrokes for text input such as form fields and prompters, and one that handles keystrokes for menu input. The latter must be able to call a menu-specific function that can determine which menu item is currently selected sp tp display a help window depending on that selection. In this way, "`F1`" can display help windows that will tell what will happen *if* you select the currently highlighted menu item.

Finally, the calling program must carefully maintain a "help state" (consisting of module and topic ID's, among other things) that will be used to select the help topic if the special help keys are ever pressed. The application must be careful to change these states whenever control passes to a new context that requires a different help topic.

This part of the help state consists of:

module – a character string containing the current module.

topic – a character string containing the current topic. This is intended to be used to display an application-specific message.

keys – a character string containing the "keys" topic. This depends on the type of input currently desired (i.e., menu selection, numeric field, etc.). This is used to display an application-independant message.

There are procedures, `h_module()`, `h_topic()`, and `h_keys()`, that the application program calls to set each of these, as well as procedures `h_save()` and `h_restore()` to save and restore the help system state. That way, an application may call `h_save()` to save the state of the help system, call `h_module()`, `h_topic()` or `h_keys()` to set the state, then call `h_restore()` to restore the state to what it was. This provides modularity.

The following table specifies how the help file topic ID is constructed from various strings in response to help request keystrokes:

Keystroke	Topic ID	Help type
`F1`	*module* + *topic*	Explain input request
`Alt-F1`	`"KEYS"` + *keys*	Show key functions
`Ctrl-F1`	*module* + `"Main"`	Help "about" module

Several functional units temporarily set the module ID when a message is being displayed. For example, when an error message is being displayed, the module is temporarily set to `ERR`, and when a data limit is being displayed, the module is temporarily set to `Lim`. In the latter case, for instance, pressing the "`Alt-L`" key when filling in a numeric field in a form will result in the limits being displayed in the status line. Pressing "`F1`" will then result in a topic from the `Lim` module being displayed.

4 Forms Package

Source files `formmeth.c`, `forms.c` and `formsio.c` provide a simple forms-package and forms i/o package, respectively.

A forms package allows the application program to "easily" define a form, using only data initialization statments, and to display and accept user input for that form easily. The forms i/o package provides routines for reading the form data from and writing the form data to files.

A form is a collection of fields of different types, all with a specified location relative to the top left hand corner of the form. When the form is displayed in a window, the fields maintain their relative locations.

The user can specify a value for each of the fields by typing in the value, for numeric and text fields, or by selecting from a menu, for choice fields. When the value for a field is completed, and the "`Enter`" key is pressed, the cursor automatically moves to the next field. The user can move ahead without entering values, by means of the "`Tab`" or "`Down-Arrow`" keys, or can move back to re-enter data at previous fields (by pressing "`Shift-Tab`" or the "`Up-Arrow`" keys.

4.1 Implementation of Fields

A field can consist of one or more elements of the same data type (e.g., floating point, integer, etc.). If there is more than one element in a field, the elements are displayed regularly spaced across and/or down the screen, and the elements are stored in a vector in the program. The usual case is to have scalar variables, and to have fields consisting of only one element. Currently, we have defined the following types of fields:

1. a Title field. This field is for providing textual captions on the form, and the field values are not editable by the user.

2. a String field. This field is for entering arbitrary text values (but on one line only).

3. an Enumerated (or choice) field. The user selects one of a small number of choices for this field. When the field is being editted, a small menu appears.

4. a Integer field. For entering integral values.

5. a Float field. For entering fractional or floating point values.

6. a Hook field. This is not for displaying or entering any data, but is intended only for allowing the application to attach functional "hooks" that will get called when the user passes from one edit field to the next. It is expected that these hooks will check the input data for validity, but in fact they may do anything.

Each field must be described by a shockingly large amount of data. There is:

- a label that gets displayed for each field,
- a"variable-name" or file id by which the field gets identified in external form files.
- a help id for the context-sensitive help system,
- its relative row and column location,
- some status flags (whether the field is hidden or not, or whether or not a value has been entered),
- a pointer to the C variable that stores the value,
- the number of displayed characters,
- the number of displayed decimals (floating point only),
- a list (array) of possible values (enumerated fields only),
- the units of the value,
- a pointer to some numerical limits, if any,
- several pointers to functions that get called to check values, set values, indicate whether a field is invisible or not, return one of the possible choices for an enum field, etc.,
- and a few more!

`forms.h` defines some macros for the various types of fields that make it a bit easier to write the initializations.

In an attempt to be extensible, the various types of fields are handled in an "object–oriented" manner. That is, the field type is indicated by a pointer to a small vector of pointers to functions. These functions are to: 1) display a field; 2) erase it; 3) allow a user to edit it; 4), 5) convert between internal and external (character) representations; 6) check the value; and 7) store the value. We have defined these seven functions for each of the six field types, all in file `formmeth.c`.

Adding a new field type will require "only" that the seven functions (or "methods") be written for that type, and that a "method table" (i.e., a vector of 7 pointers to those functions) be initialized, and that a pointer to that vector be placed in all new fields of the new type.

4.2 Implementation of Forms

A form is implemented as simply a one-dimensional array of fields. The algorithms for displaying and editing of the fields within a form are pretty simple. We start at the first field, and visit each field in turn. If the field is a title field, we simply display some text. If the field is a hook field, we call the check function, if defined. Otherwise, we visit each element of the field in turn, displaying the value and letting the user edit that value.

Special keys, such as "`Tab`" (or "`Down`") or "`Shift-Tab`" (or "`Up`") cause the current cursor to move down or up without changing a value. When the cursor moves down, it moves to the next element of a field, or to the next field if we have visited all elements. When the cursor moves up, it moves to the previous element of a field, or to the previous field if we are at the first element.

4.3 Input/Output of Forms

The `formsio.c` source file defines a module for the writing and reading of forms. When a form is written to a file, it is in pure text format, and the first few lines of the file define the name of the form and the version of the forms package it was generated on.

To write a form to a file is simple – just process each *non*-title, *non*-hook field in order, writing the file ID of each field before the value (or list of values, in the case of vectors longer than 1). For the output of vectors, the subscript range is also output in square brackets, `[]`, after the file id, (or variable name).

When reading a form file, options allow the generation of error messages if the file contains a field not in the form, or conversely, allow such fields to be ignored without comment. In a form file, the fields do not have to be in any particular order, but the ID and all data values must be on one line.

The following is an extract from the configuration file, `pcl.cfg`, which in fact is implemented as a form file:

```
##Form: CFG
##FormsVersion: 1.0
##Time:  Tuesday, August 21, 1990; 11:45:10 AM

;; PCL Configuration file
;; This file provides default values for many parameters.
;; Maximum speed of vehicles (cars and trucks) on the highway

Maximum-Speed: 90.0

;; Default design speed of the highway

Design-Speed: 100.0
```

Here we see that the name of the form is `CFG` (programs must supply this name when attempting to input the file), that blank lines and comments are allowed, and that two scalar variables are defined (`Maximum-Speed` and `Design-Speed`) – both floating point values.

Following is a *portion* of the data in a site file:

```
##Form: RTC
##FormsVersion: 1.0
##Time: Sun Aug 19 21:13:14 1990

Site-ID: example1
Description: Example problem from final report
Distance[0..2]:     0  1000  2000
Grade[0..1]:   7.0   0.0
Use-Average-Grade: No
Average-Grade:?
Passing-Allowed: Yes
Percent-No-Passing: 0.0
DHV-Percent-of-AADT: 17.4
DHV[0..3]:   300   500  1000  1500
Percent-Upgrade: 50.0
Percent-Trucks: 10.0
Percent-RVs: 10.0
Peak-Hourly-Factor:  1.0
```

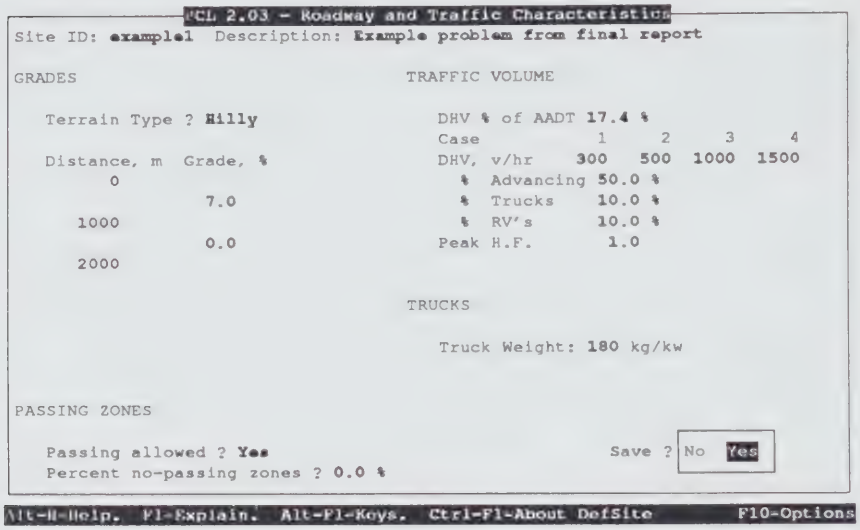

Figure 4.1: Roadway and Traffic Characteristics Form

```
Truck-Weight: 180
Module-Site-Definition:    1
```

This one has form name RTC and this portion defines 16 variables. The first two (Site-ID and Description) are character strings (one string each). The second two (Distance and Grade) are floating point vectors of three and two elements, respectively.

The form that that resulted in the above site description file is shown as it is displayed on the screen in Figure 4.1.

A portion of the *C* program that defined that form is as follows:

```
FORM    f_tab = {
        F_S( sitep, "Site ID: ", "Site-ID", "SiteID",
                0,  0, F_DEF|F_LOW|F_DELW, sizeof(site) ),
        F_S( descp, "Description: ", "Description", "Description",
                0, 19, F_DEF, sizeof(description) ),
        F_T( "GRADES", 2, 0, 0 ),
        F_E( terrain, "Terrain Type ? ", "Terrain-Type", "Terrain",
                4, 3, F_DEF, 7, tt_names, NULL ),
        F_HV( "Classify-Terrain", 0, terr_check ),
        F_E( laneType, "Lane Type ? ", "Lane-Type", "LaneType",
                4, 3, F_DEF|F_HIDE, 7, lt_names, NULL ),
        F_TI( "Distance, m  Grade, %", "TDistGrade", 6, 3, F_HIDE ),
```

```
        F_TI( "Site Length, m", "TSiteLen", 6, 3, F_HIDE ),
        F_F( siteLen, "", "Site-Length", "SiteLen",
                7, 5, F_DEF|F_HIDE, NULL, 6, 0 ),
        F_FV( dist, "", "Distance", "Grades-D",
                7, 5, F_RJ|F_HIDE, NULL, 5, 0, 5, 2, 0 ),
        F_FV( grade, "", "Grade", "Grades-G",
                8, 16, F_RJ|F_HIDE, NULL, 5, 1, 4, 2, 0 ),
        F_HV( "Grade-Check", 0, grade_check ),  /* data checker */
        F_ESH( use_avg_grade, "Use Avg Grade","Use-Average-Grade","AvgGrade",
                15, 3, F_DEF|F_HIDE, 3, no_yes, NULL, use_avg, check_avg ),
        F_F( avg_grade, "", "Average-Grade", "", 15, 3, F_HIDE, NULL, 5, 0 ),
        F_T( "PASSING ZONES", 18, 0, 0 ),
        F_ES( passing, "Passing allowed ? ", "Passing-Allowed", "Passing",
                20, 3, F_DEF, 3, no_yes, NULL, pass_fn ),
        F_F( fract_nop,"Percent no-passing zones ? ","Percent-No-Passing",
                "PercNoPass", 21, 3, F_HIDE, " %", 3, 1 ),
        F_T( "TRAFFIC VOLUME", 2, 37, 0 ),
        F_F( dhv_percent, "DHV % of AADT ", "DHV-Percent-of-AADT", "DHVPerc",
                4, 40, F_RJ|F_DEF, " %", 4, 1 ),
        F_T( "Case           1     2     3     4", 5, 40, 0 ),
        F_IV( dhv,    "DHV, v/hr  ", "DHV", "DHV",
                6, 40, F_RJ, NULL, 5, 4, 0, 6 ),
        F_F( p_adv[0], "  %  Advancing ", "Percent-Advancing", "PercAdv",
                7, 40, F_RJ, " %", 4, 1 ),
        F_F( p_tr[0], "  %  Trucks    ", "Percent-Trucks", "PercTrucks",
                8, 40, F_RJ, " %", 4, 1 ),
        F_F( p_rv[0], "  %  RV's      ", "Percent-RVs", "PercRVs",
                9, 40, F_RJ, " %", 4, 1 ),
        F_HV( "Percent-Check", 0, percent_check ),      /* data checker */
        F_F( phf[0],     "Peak H.F.      ", "Peak-Hourly-Factor", "PHF",
                10, 40, F_RJ|F_DEF, NULL, 4, 1 ),
        F_TI( "TRUCKS", "TTruckWeight", 13, 37, F_HIDE ),
        F_E( truck[0], "Truck Weight: ", "Truck-Weight", "TruckWeight",
                15, 40, F_DEF|F_HIDE, 3, weights, " kg/kw" ),
        F_ES( saving, "Save ? ", "Saving", "Save",
                20, 56, F_DEF|F_NOF, 3, no_yes, NULL, save_fn ),
        F_I( m_sitedefn, NULL, "Module-Site-Definition", "MSiteDefn",
                21, 56, F_DEF|F_HIDE, NULL, 4 ),
        };
int     f_tab_len = sizeof(f_tab)/sizeof(f_tab[0]);
```

To display the form and let the user edit it requires simply a call to `form_do()`:

```
        form_do( win, 0, 0, f_tab, f_tab_len, &Changed );
```

To write the form to the output form file simply requires a call to `form_out()`:

```
if( form_out( f_tab, f_tab_len, "RTC", Outfile, 0 ) == 0 )
        Changed = 0;
```

5 Miscellaneous Utilities

5.1 Screen Colours

Source file `colours.c` supplies all of the logic needed to maintain various text and border colours on colour displays (and on monochrome displays). Basically, there are about 22 different types of text and/or border and highlighting effects defined. Tables in that file give the foreground and background colours to use for each type.

To change the colour of a particular portion of the display, locate the corresponding item in the tables, and change the two colours to your preference.

Part III

Application Details

6 Speed–Distance Calculations

6.1 Speed–Distance Tables

In order to compute the start and end locations of the climbing lane, based on the speed drop of a heavily loaded truck, a set of speed-distance tables must be provided for each weight of truck. The default tables distributed with PCL are for truck weights of 120 kg/kw and 180 kg/kw.

For each weight of truck, three tables must be supplied, each in a separate data file. The tables must give speed-distance data for acceleration downgrade, acceleration upgrade, and deceleration upgrade.

The following file names must be used:

`m2t`*nnn*`.ad`	– for acceleration downgrade.
`m2t`*nnn*`.au`	– for acceleration upgrade.
`m2t`*nnn*`.du`	– for deceleration upgrade.

where:

nnn is the truck weight, in kg/kw.

The acceleration tables must start at a speed of 0 km/hr and contain data for increasing speeds up to the truck speed given in the configuration file, or to the maximum speed attainable on the grade, whichever is least.

The deceleration table must start at a speed at least as high as the truck speed given in the configuration file and must contain data for decreasing speeds down to the maximum speed attainable on the grade, or until a distance of 3000m has been covered.

The upgrade tables must contain data for grades of 0%, 1%, ... 10%, in increments of 1%. The first 2 data columns give speed and distance for 0%, the second two for 1%, and so on.

The downgrade tables must contain data for grades of –10%, –9%, ... –1%, in increments of 1%. The first 2 data columns give speed and distance for –10%, the second two for –9%, and so on.

Each column is exactly 9 characters wide.

The first line of every file must contain a description of the data in the file, as the following example line from file "`m2t120.ad`" shows (note that due to width limitations here, the line is shown in two parts):

```
#SDT M2   Truck     120 kg/kw  90 km/h  AccelDown -10%  -1%
Speed-Dist 90/02/10
```

Columns	Contents
1–4	`#SDT`
6–7	`M2`
11–15	`Truck`
21–23	truck weight
25–29	`kg/kw`
31–33	truck speed
35–38	`km/h`
41–45	`Accel` or `Decel`
46–49	`Up` or `Down`
51–53	percent grade of first curve
54	`%`
56–58	percent grade of last curve
59	`%`
61–70	`Speed-Dist`
72–79	date

6.2 Basic Algorithm

Source file `cl_comp.c` contains all the logic for calculating speed versus distance travelled of a truck on grade, and for computing the location of climbing lanes based on that.

6.2.1 Basic Terminology

A section is that portion of roadway that has a constant grade; there are typically several sections in a site. The length of each section is the difference between the distances at each end of the section, as defined in the `defsite` form (see Figure 4.1, for example).

A zone is the portion of roadway from the start to the end of a continuous passing lane. Any given site may have more than one zone (or none), and any zone may partially or completely overlap one or more sections.

A grade is simply the physical characteristics of an abstract grade (i.e., those characteristics that are applicable to many sections of the same

grade). This includes the slope, in percent, and the (possibily interpolated) speed-distance tables for that grade.

A roadway that is characterized by the following data:

Distance, m	**Grade**, %
0	
	3.2
1000	
	-2
2000	
	3.2
3000	

has 3 sections and 2 different grades.

6.2.2 Preparation of Tables

The site data is examined, and a list is made of all different grades. The speed distance tables are read, and only the data necessary for the actual grades are loaded. If a grade is integral, the data can be used directly from the table. If a grade is fractional, we interpolate speed-distance from the next lower and higher integral grades.

For positive grades, we load both acceleration and deceleration tables; for negative grades we load only the acceleration tables. For positive (uphill) grades, we must also determine the maximum possible truck speed – this is the assymtote of the tables.

In the above example, we had two different grades of -2% and +3.2%. For this case, we would load acceleration speed-distance tables for -2%, and acceleration and deceleration tables for +3% and +4%, then interpolate those for 3.2%. For the +3.2% grade, we would also search the acceleration and deceleration tables to find the maximum possible truck speed.

6.2.3 Calculation of Speed Along Grade

We wish to construct a table of truck speed versus distance from datum.[1] The basic method of predicting the speed of the truck as it travels along the site is as follows:

1. Set the entry speed of the first section to be the assummed truck speed.

2. For every section in the site, do the following:

[1] Datum is the "0" point from which the distances are measured.

(a) If the grade is positive, and the entry speed is greater than the maximum truck speed for that grade, select the deceleration table, else select the acceleration table.

(b) Find the distance in the table at which the speed equals the entry speed (interpolate if necessary). Call this distance the *offset*.

(c) For each entry in the selected speed-distance table after that, do the following:

   i. Subtract the *offset* from the distance in the table; this gives the distance from the start of the section at which the truck speed equals the speed in the table. From this, add one entry to the table of speed versus distance from datum.

   ii. Stop when the distance exceeds the length of the section, interpolating between the last two table values, if necessary.

(d) Set the last speed of the section to be the entry speed of the next section (if there is a next section).

6.2.4 Calculation of Lane Start and End

The speed-distance tables for the site that were constructed above are scanned for all regions in which the truck speed drop exceeds the allowable truck speed drop.

7 PCL Files

7.1 Configuration File

The configuration file, **pcl.cfg**, is used to provide default values for many parameters that rarely change. See Sections 3.1.1 and 3.2 of the User's Manual for more information about the various parameters that can be set.

The configuration file is maintained as a form file (see section 4.3), and is loaded at the start of every module's execution.

A *typical* configuration file is shown below (although yours may not match this exactly):

```
##Form: CFG
##FormsVersion: 1.0
##Time:  Tuesday, August 21, 1990; 11:45:10 AM

;; PCL Configuration file
;; This file provides default values for many parameters.
;; Many of these parameters can also be changed by the user at run time,
;; some through DOS environment variables, and others through direct
;; input in a screen-form.  See the User's Manual (Chapter 3) for
;; more information.

;; Maximum speed of vehicles (cars and trucks) on the highway

Maximum-Speed: 90.0

;; Default design speed of the highway

Design-Speed: 100.0

;; Maximum truck speed for speed-distance calculations on grade.
;; The speed-distance tables must contain data for at least this speed.
;; See Chapter 3 in the User's Manual

Truck-Speed: 90.0
```

```
;; truck speed drop to use to calculate start and end positions of lane

Start-Speed-Drop: 15.0
;;End-Speed-Drop: 15.0

;; Possible truck weights (in kg/kw) to use for the speed-distance
;; calculations.  Speed-distance tables must exist for each of these.
;; See Chapter 3 in the User's Manual

Truck-Weights: 120 180

;; Climbing lane parameters.
;; Transition distances are currently not used.

Start-Transition-Distance: 200.0
;;End-Transition-Distance: 200.0
Minimum-Lane-Length: 1500.0

DHV-Percent-of-AADT: 17.4

;; the peak hourly factor can be less than 1.0 to account for significant
;; peaking within the hour.  See HCM (1985).

Peak-Hourly-Factor: 1.0

;; paths and locations of various files
;; directory containing executables
PCL-Path: \work\pcl
;; directory - speed-distance files and other tables
PCL-Tables: \work\pcl\tables
;; help database file
PCL-Help: \work\pcl\pclhlp.hlp
;; directory - Borland Graphics Interface (BGI) files
BGI-Path: \work\pcl\bgi;\turbo\bgi;e:\turbo\tc\bgi

Help-Level: Novice
Base-Year: 1989
Real-Rate-of-Return: 5.0
Inflation-Rate: 5.0
Life-of-Facility: 30
Vehicle-Hour-Cost: 5.0
Vehicle-Accident-Cost: 10000.0

;; total lane cost = Lane-Fixed-Cost + length*Lane-Unit-Cost
```

```
Lane-Unit-Cost: 396000.0
Lane-Fixed-Cost: 0.0
Lane-Maintenance-Cost: 0.0
```

The installation procedure modifies the distributed version of this file. In particular, some of the search paths and file names are modified to reflect the actual installed locations.

Note that if PCL can find and load this file, this file gives the locations of all other required files. No error message is produced if this file is not loaded. However, the configuration file normally resides in the same directory as the executables, and thus will be loaded properly. See Section 3.1.1 of the User's Manual for a description of the obsessive method PCL uses to find the configuration file.

Any given PCL module will require only a subset of the values in the file, so the file is loaded in a mode that ignores extra variables.

7.2 Communication File

The main PCL program loads and executes several other secondary executable programs (**defsite**, **defcl**, **estperf**, **effmeas**). When these programs are executed, command line arguments are supplied to tell them what site to work on (i.e., which site file to load).

During execution of a secondary module, the user may switch to a different site file, and this change must be communicated back to PCL.

Thus, each secondary module writes a communication file just before it terminates. The communication file contains three lines giving the site ID, site description, and site file name.

The communication file is a temporay file whose name is created by PCL and given to the secondary module via a command-line argument.

7.3 Speed–Distance Tables

The speed–distance tables are tables of truck speed versus distance on diiferent grades. They are documented in Section 6.1 of this document, and in Section 3.3 of the User's Manual.

7.3.1 Higher Truck Speeds

These are the modifications that must be made in order to use truck speeds of higher than the current limit (90 km/h).

1. Generate three new speed–distance table files for each truck weight, containing speeds at least as high as the new limit. The format of these tables has to be exactly as documented in Section 6.1.

   This is the hard part. Contact the authors of this document for information about how to do this.

2. Install the files in the `tables` sub-directory (i.e., in the directory given as the value of the `PCL-Tables` variable in the configuration file).

3. Modify the `Truck-Speed` variable in the configuration file to use the new speed (see Section 7.1).

7.3.2 Other Truck Weights

These are the modifications that must be made in order to use truck weights of other than the 120 and 180 kg/kw supplied with PCL.

1. Generate three new speed–distance table files for the new truck weights. The format of these tables has to be exactly as documented in Section 6.1, and they have to be named according to the convention described there.

   This is the hard part. Contact the authors of this document for information about how to do this.

2. Install the files in the `tables` sub-directory (i.e., in the directory given as the value of the `PCL-Tables` variable in the configuration file).

3. Modify the `Truck-Weights` variable in the configuration file to also contain the new weight. (see Section 7.1).

7.4 Coefficients File

In order to predict the performance of a passing lane, a large number of simulations were performed and a number of regression equations were developed. These equations relate average speeds and percent platooning to a number of different variables (nine in all), for four different cases.

The equations are evaluated by summing the products of a set of coefficients and their associated variables. The coefficients are specified in a data file, normally called `plcoeff.dat`.

The file structure is fairly simple. Blank lines and lines beginning with ";" are treated as comments, and ignored. The remaining lines are in 10 columns each, each column 16 characters wide. The first column of each

line contains a label describing which case the data in the rest of the line is for. The remaining columns contain numbers, with a blank column taken as zero (0).

The numbers in each column are multipliers of the following quantities (with the exception of column 2, which is a constant):

Col.	Posn.	Quantity to be multiplied
2	17 – 32	1
3	33 – 48	Two Way Volume, veh/hr
4	49 – 64	Direction 1 Volume, veh/hr
5	65 – 80	Direction 2 Volume, veh/hr
6	81 – 96	Percent trucks, %
7	97 – 112	Percent RV's, %
8	113 – 128	Natural log (ln) of the effective distance, m
9	129 – 144	Natural log (ln) of the lane length, m
10	145 – 160	Direction 1 volume, % of total

The labels in column 1 are made up by catenating the following strings, in the order shown.

`n`	*or*	*nothing*	– no passing lane *or* passing lane
`l`	*or*	`r`	– level *or* rolling terrain
`pa`	*or*	`pna`	– passing allowed *or* not allowed
nothing	*or*	`h`	– low *or* high platooning
nothing			– equation applies to both directions
`1`	*or*	`2`	– applies to direction 1 (advancing) *or* 2

For example, the label `nlpna` marks the regression equation on that line as being for no passing lane, level terrain, passing not allowed, low platooning, and applying to both directions.

The data file is organized into 4 sections, as follows:

1. 8 lines specifying the equations for percent platooning, without passing lane.

2. 8 lines specifying the equations for percent platooning, with passing lane.

3. 12 lines specifying the equations for average speed, without passing lane.

4. 12 lines specifying the equations for average speed, with passing lane.

The sections must be given in the order shown. Within each section (each of which covers all necessary combinations of level/rolling terrain, passing allowed/not allowed, low/high platooning, and sometimes direction 1/direction 2), the lines must also be in the order:

```
nlpa
nlpah
nrpa
nrpah
nlpna
nlpnah
nrpna
nrpnah
```

for the percent platooning equations. The order must be:

```
nlpa
nlpah1
nlpah2
nrpa
nrpah1
nrpah2
nlpna
nlpnah1
nlpnah2
nrpna
nrpnah1
nrpnah2
```

for the average speed equations. In other words, do not change the order of the lines in the distributed version of **plcoeff.dat**.

Part IV

Appendices

8 List of Functions

The following is an alphabetic list of almost all of the functions defined in the PCL source files, for program version 2.03. Not included are macros, nor any of the Blaise routines, nor any standard C routines.

Caution: there may be duplicates of some functions.

`Abort()` (`err.c`) Print a fatal error msg to error output, and die.

`BGI_init()` (`graph.c`) Initialize the graphics system. Locate the Borland BGI files and initialize with device autodetection. Determine device coordinate ranges, and set the transformations to identity.

`BGI_lineto()` (`graph.c`) Move the "pen" to the given position, drawing a straight line from the current position to the given one.

`BGI_moveto()` (`graph.c`) Move the graphics "pen" to the given position, without drawing a line.

`BGI_rect()` (`graph.c`) Draw a filled rectangle, given lower left and upper right coordinates. "pattern" specifies the fill pattern to use (currently, only a slash is permitted).

`BGI_setfont()` (`graph.c`) Set the current font parmeters to those given, including direction and size. Currently, we only support a sans serif font, in horizontal or vertical direction.

`BGI_setls()` (`graph.c`) Set line style and thickness.

`BGI_term()` (`graph.c`) terminate graphics system. Reset transformations, prompt user to strike a key and wait for it. Clear graphics screen and restore screen to text mode.

`BGI_text()` (`graph.c`)

`Error()` (`err.c`) Display an error message, and return to caller.

`Fatal()` (`err.c`) Display a fatal error message, and terminate execution.

`LConfirm()` (`err.c`) Print a warning message about a data limit violation, then ask the user to confirm that the value is O.K.

`LError()` (`err.c`) Display a error message about a data limit violation, and return to caller.

`LWarn()` (`err.c`) Print a warning message about a data limit violation, and return to caller.

`PS_flush()` (`graph.c`) Ensure that the currently active Postscript path is stroked and emptied.

`PS_init()` (`graph.c`) Initialize the graphics system. Determine device coordinate ranges, and set the transformations to identity.

`PS_lineto()` (`graph.c`) Move the "pen" to the given position, drawing a straight line from the current position to the given one.

`PS_moveto()` (`graph.c`) Move the graphics "pen" to the given position, without drawing a line.

`PS_rect()` (`graph.c`) Draw a filled rectangle, given lower left and upper right coordinates. "pattern" specifies the fill pattern to use (currently, only a slash is permitted).

`PS_setfont()` (`graph.c`) Set the current font parmeters to those given, including direction and size. Currently, we only support a sans serif font, in horizontal or vertical direction.

`PS_setls()` (`graph.c`) Set line style and thickness.

`PS_term()` (`graph.c`) terminate graphics system. Clear graphics screen and restore screen to text mode.

`PS_text()` (`graph.c`)

`PV()` (`estpass.c`) print a vector of floating point values in the current window.

`TLV()` (`estpass.c`) print a vector of strings in the log file.

`TPV()` (`estpass.c`) print a vector of floating point numbers in the log file.

`WTLV()` (`estpass.c`) print a vector of strings in the current window.

`WTPV()` (`estpass.c`) print a vector of floating point numbers in the current window.

`Warn()` (`err.c`) Print a warning message, and return to caller.

`XX_init()` (`graph.c`) Initialize the graphics system. Determine device coordinate ranges, and set the transformations to identity.

`XX_lineto()` (`graph.c`) Move the "pen" to the given position, drawing a straight line from the current position to the given one.

`XX_moveto()` (`graph.c`) Move the graphics "pen" to the given position, without drawing a line.

`XX_rect()` (`graph.c`) Draw a filled rectangle, given lower left and upper right coordinates. "pattern" specifies the fill pattern to use (currently, only a slash is permitted).

`XX_setfont()` (`graph.c`) Set the current font parmeters to those given, including direction and size. Currently, we only support a sans serif font, in horizontal or vertical direction.

`XX_setls()` (`graph.c`) Set line style and thickness.

`XX_term()` (`graph.c`) terminate graphics system. Clear graphics screen and restore screen to text mode.

`XX_text()` (`graph.c`)

`a_Er()` (`tables.c`) Return the "Er" factor (passenger-car equivalent for RV's) for the given level of service index, approach.

`a_Et()` (`tables.c`) Return the "Et" factor (passenger-car equivalent for trucks) for the given level of service index, approach.

`a_avgs()` (`tables.c`) Return the average speed for the given level of service index and design speed, approach.

`a_fd()` (`tables.c`) Return the fd value for a given % in uphill direction, approach.

`a_fw()` (`tables.c`) Return the fw factor for the given level of service index, lane width index, and shoulder width index.

`a_ilos()` (`tables.c`) Return the level of service index for the given average speed and design speed, approach.

`a_vcr()` (`tables.c`) For a given level of service index, % no passing, design speed, return the v/c ratio from the table and the average speed corresponding to the level of service for approach.

`add_history()` (`m_util.c`) Add a menu and its just selected choice to the history list.

`adjust_zones()` (`defcl.c`) Adjust the computed zones of climbing lane by adding starting and ending transition distances (if appropriate), increasing them to minimum length, then merging overlapping or close zones. All this because "compute_sl()" may compute more than one zone of lane (esp. if there is a down grade in the middle of the site).

`air()` (`estperf.c`) Calculate accident involvement rate from the given speed reduction.

`alphasort()` (`scandir.c`) Compare file names alphabetically, for sorting.

`bfopen()` (`cl_comp.c`) Open a file in binary mode. Return the file pointer.

`browse_modules()` (`helpsys.c`) Repeatedly show the module selection menu until user selects "Exit".

`buf_edit()` (`formmeth.c`) display and let the user edit the characters for the given field. Provide the given default value. Return an indication of whether edit was OK, whether field was changed, and whether to move up or down for next field.

`build_mask()` (`cl_comp.c`) Initialize the speed-distance curves for the various grades, and build a mask specifying which columns (grades) must be loaded from the data files. load_mask[i] != 0 iff column i must be loaded.

`build_sections()` (`cl_comp.c`) From the input distance and grade arrays, check the data for saneness and construct the required global data structures. Create a SECTION_DATA record for each section of constant grade along the roadway. For each different numerical value of grade found, start a new GRADE_TABLE record.

`build_table()` (`cl_comp.c`) Build the complete speed-distance tables for the given grade, and complete the GRADE_TABLE. Positive grades have both acceleration and deceleration curves; negative grades have only deceleration. Interpolate the curves, if necessary.

`build_tables()` (`cl_comp.c`) Build all of the grade tables.

`calc()` (`estpass.c`) Calculate all performance data, and print results on the screen.

`calc()` (`estperf.c`) Calculate all performance data, and print results on the screen.

`calcPPAS()` (`estpass.c`) calculate percent platooning and average speed for one case. Use the regression equations developed by Khan et. al.

`calc_accidents()` (`estperf.c`) Calculate accident involvement rates and numbers for the various DHV's, for the case of no climbing and climbin lanes. Calculate the reduction expected from adding climbing lane. Log intermediate results, if requested.

`calc_all()` (`effmeas.c`) Get user input of parameters, then calculate and print all effectiveness measures, benefits, and cost effectiveness.

`calc_all()` (`estpass.c`) Display input form, get user input, then calculate all results.

`calc_all()` (`estperf.c`) Display input form, get user input, the calculate all results.

`calc_approach()` (`estperf.c`) Calculate approach speeds and levels of service for the various DHV cases. Log intermediate results, if requested.

`calc_ben()` (`effmeas.c`) Calculate the gross and net benefits from adding a climbing lane. Print results on the screen.

`calc_ceff()` (`effmeas.c`) Calculate the cost-effectiveness measures. Print results on the screen.

`calc_cost()` (`effmeas.c`) Calculate the total yearly cost of adding the climbing lane. Print results on the screen.

`calc_delay()` (`estperf.c`) Calculate delay reduction due to effects of adding climbing lane. Log intermediate results, if requested.

`calc_lane()` (`estperf.c`) Calculate average upgrade speed for the case of climbing lane. Use regression equations to do so. Print intermediate results, if requested. It is expected that the average grade over the site will be used.

`calc_mem()` (`effmeas.c`) Calculate multiple effectiveness measures. Compute relative values of delay and accident reductions, then compute the weighted combination. Print results on screen.

`calc_sem()` (`effmeas.c`) Calculate single effectiveness measures. That is, calculate delay reduction in min/100 veh, and accident reduction in accidents/million veh-km. Print to screen, and to log file (if requested).

`calc_upgrade()` (`estperf.c`) Calculate the upgrade speed and Level Of Service, for the case of no climbing lane. Log intermediate results, if requested. It is expected that the average grade over the whole site will be used.

`check_avg()` (`defsite.c`) Check to see if an average grade can be used. Base this on the HCM (1985). If it is possible that an avg grade can be used, add the question to the form (by changing the label), and return 1 to indicated the the field should not be hidden. This is called whenever the grades and distances have been filled in.

`check_field()` (`forms.c`) check the supplied value for the ith element of given field. First check the limits, if any. Then call the user's verify function, if it exists. Change the F_EDOK bit in the supplied return code, depending on if the value is O.K. or not.

`check_file()` (`cl_comp.c`) Checks the few characters of data in an opened file to see if it looks like a Speed-Distance table file. The first 4 characters are a tag and must be "#SDT" (for text) or "#SDB" (for binary). In either case, the first (header) line is read. For binary files, the width of each row of the file is determined (from the grade info in the header line).

`check_passing()` (`estpass.c`) Check the site for passing opportunities. Display a table of percent passing opportunities, and flag those that do not meet the criteria.

`choice_edit()` (`formmeth.c`) display a horizontal choice menu for an enumerated field and allow user to select one of the choices. Provide a default choice, and return indication of whether value was changed, and whether to move up or down for next.

`choose()` (`helpsys.c`) Pop up the main help system menu, and dispatch on the users selection, displaying either the selected help topics, or popping up more menus.

`choose_module()` (`helpsys.c`) Pop up the module choices menu, and display the main help topic for the chosen module.

`cl_draw()` (`cl_draw.c`) Draw all of the intersting plots (elev vs distance, speed vs distance, and climbing lane zones). Determine max and min of the various values so that we can autoscale to fit the screen.

`coef()` (`estpass.c`) calculate a regression equation, given input vectors of coefficients and variables.

`colour_attr()` (`colours.c`) Return the colour attribute byte for the given text type. The attribute byte is specific to IBM PC's and compatibles.

`colour_back()` (`colours.c`) Return the colour to use for the background of the given text type.

`colour_find()` (`colours.c`) return a pointer to the colour descriptor for the given text type. Initialize the colour tables first, if necessary. Return the default colour for unknown text types.

`colour_fore()` (`colours.c`) Return the colour to use for the foreground of the given text type.

`colour_init()` (`colours.c`) Initialize the table of colours for the various text types. If the CRT type is not specified, try to automatically detect the type of CRT.

`compute_1sd()` (`cl_comp.c`) Compute the speed distance curve for one section of the site.

`compute_cl()` (`cl_comp.c`) Compute start and end of all zones of climbing lane. They start where speed drops to "start_speed", and end where speed rises back to "end_speed", A site can have many zones.

This is implemented as a state engine, with two states: 0 = looking for a start point, and 2 = looking for an end point. We look for two speeds that bracket the speed we are looking for, then interpolate the distance between.

`compute_it()` (`defcl.c`) Compute the climbing lane locations (zones) depending on truck speed drop.

`compute_sd()` (`cl_comp.c`) compute the speed-distance curves for all sections, setting the entry_speed for all but the first to be the exit speed of the previous section.

`cr_prompter()` (`w_util.c`) Create and display a prompter window of given size, with geiven label.

`crt_restore()` (`crt.c`) Restore the state of the CRT to that saved earlier. If requested, free the memory that was allocated to hold the screen text (this implies that the same screen cannot be restored again).

`crt_save()` (`crt.c`) Svae the current state of the CRT. This includes all the visible text and the position of the cursor.

`dhv_label()` (`effmeas.c`) function to prepare the label for one of the points on the graph.

`display_field()` (`forms.c`) display the i'th element of a field. "vflag" is 1 to force a value display, 0 to have value displayed only if it has been set or is a default value.

`do_browse()` (`helpsys.c`) Repeatedly show the main help menu until user selects "Exit".

`do_dos()` (`help.c`) Escape temporarily to DOS. Save and restore screen.

`do_dos()` (`pcl.c`) User wishes to exit to DOS. Save current screen and restore to what it was at start of run. Exit to DOS, restoring screen on return.

`do_err()` (`err.c`) If possible, display error message in status line, otherwise print it to standard error output.

`do_exec()` (`pcl.c`) execute the given program module. Search for the executable in the PCLPath. Create a temp file for use as a communication file (it may be written by the module). The temp file name is given to the module via command-line argument. If the CRT type is known, inform the module of that fact. Write message in status line, execute the module, then read the com file.

`do_fileops()` (`pcl.c`) Handle the site/file operations module. Present the File Operations menu, and interact with the user accordingly, returning when "Exit" chosen.

`do_help()` (`help.c`) Display a help window. Topic is located by concatenating module id and topic id.

`do_interact()` (`defcl.c`) Create the main screen window, initialize the help system, read the configuration and site files, and enter main user interaction loop.

`do_interact()` (`defsite.c`) Create main window, initialize the help system, read config and site files (if appropriate) then interact with user.

`do_interact()` (`effmeas.c`) Load all data files, and interact with user. Create a virtual window to hold the output. Initialize the help system. Load configuration and site files.

`do_interact()` (`estpass.c`) Create the main window for output, initalize help system, read data files, interact with user.

`do_interact()` (`estperf.c`) Create the main window for output, initalize help system, read data files, interact with user.

`do_it()` (`defcl.c`) The main user interaction loop. Print the header data describing the site. Display the user menu and dispatch on the selection made.

`do_it()` (`effmeas.c`) Set limits to input data, get input values from user, calculate all results, then loop presenting menu to user, until she signals "Quit".

`do_it()` (`estpass.c`) Calculate all results, then repeatedly display menu and interact with user.

`do_it()` (`estperf.c`) Calculate all results, then repeatedly display menu and interact with user.

`do_menu()` (`effmeas.c`) Display menu, read selection, and dispatch on the selection.

`do_menu()` (`estpass.c`) Display menu, read selection, dispatch on selection.

`do_menu()` (`estperf.c`) Display menu, read selection, dispatch on selection.

`do_menu()` (`help.c`) Display and handle the F10 options menu. This is the default options menu function; it can be replaced by an applications function. If the application has not defined an exit function, protect that menu item.

`do_new()` (`pcl.c`) User has elected to specify a new site. Prompt for ID, description, and site file name. Write ID and description to site file.

`do_open()` (`pcl.c`) The user wishes to select an existing site. Prompt for the site file, with a default answer being all those with extension ".rtc". Allow user to select file from a menu if she elects to browse. Allow user to specify a NULL file. If file is not NULL, read the file to get important info (ID and description).

`do_sdump()` (`help.c`) Save a screen dump (contents plus attributes) in a file. Prompt user for file name, after providing a suitable default. Save name for default next time.

`doll()` (`effmeas.c`) Given a format, return a formatted string containing the baseyear. This intended for use in printing the units of the various output lines.

`draw_grades()` (`cl_draw.c`) Draw the grade profile (i.e., elevation versus distance) on the top portion of the display.

`draw_speeds()` (`cl_draw.c`) Draw the variation of speed versus distance on the lower portion of the screen. Also draw the truck speed below which a climbing lane is advised.

`draw_zones()` (`cl_draw.c`) Show the zones chosen for the climbing lane.

`ed_prompter()` (`w_util.c`) Allow the user to edit the text within a prompter window. Supply default value.

`edit_field()` (`forms.c`) edit the i'th element of a field.

`enum_check()` (`formmeth.c`) Check the enum value for validity.

`enum_display()` (`formmeth.c`) Display an element for an enumeration field

`enum_edit()` (`formmeth.c`) edit a vlaue for an enumeration field.

`enum_erase()` (`formmeth.c`) erase an element for an enumeration field.

`enum_store()` (`formmeth.c`) store the integer (or string) value for the i'th element of an enum field.

`enum_string()` (`formmeth.c`) return the characters for an element for an enumeration field.

`enum_value()` (`formmeth.c`) convert characters to internal value (index) for an element of an enumeration field.

`erase_field()` (`forms.c`) erase the i'th element of a field.

`f_chngfield()` (`forms.c`) Set the GOTVALUE bit in the flags of the field referencing the given variable. This indicates to the forms package that the variable has a value, presumably set by some mechanism outside the package. Display all elements of the field. This allows the application program to change the value of a field variable, and have the displayed form updated.

`f_choice()` (`formmeth.c`) Return the character string for the i'th choice for an enumerated field. Use the array of choices, or the choice function if that does not exist.

`f_compare()` (`f_util.c`) Sort comparison function for file names. File names come before directory names. Names of like type are sorted alphabetically, except ".." always appears last.

`f_curform()` (`forms.c`) establish the given form as the currently active form.

`f_deflimf()` (`forms.c`) define a verification function and/or the full set of limits for a floating point field variable.

`f_deflimi()` (`forms.c`) define a verification function and/or the full set of limits for an integer field variable.

`f_defvar()` (`forms.c`) Set the GOTVALUE bit in the flags of the field referencing the given variable. This specifies to the forms package that the variable has a value.

`f_defvf()` (`forms.c`) define a verification function for the given field variable.

`f_execdir()` (`f_util.c`) extract the directory from the pathname of the executable file "exec_file". DOS fills this in and provides it as argv[0] to programs. UNIX does not fill in the directory, and thus this function should probably (but doesn't) search the UNIX PATH.

`f_exists()` (`f_util.c`) return TRUE iff file of given name exists.

`f_filter()` (`f_util.c`) Function to select (filter) names from a directory. Selected names are directories (except ".") and names that match the current pattern. This function selects the names that will appear in the file selection menus.

`f_find()` (`f_util.c`) Try our damndest to find a file; be ridiculously obsessive about it. First try the name given by "name"; If not successful, try the value of environment variable "name_ev"; If not successful, search the path given by environment variable "path_ev"; If

not successful, search the path given by "path"; If not successful, try in the directory given by global variable "Exedir". Return full, after filename has been copied to it. If file not found, return NULL.

f_gotval() (**forms.c**) return TRUE if and only if the forms package thinks that the variable has a value, indicated by the GOTVALUE bit in the flags. This bit is set when a user edits the value on the screen, or when an application calls "f_defvar" or "f_chngfield".

f_isdir() (**f_util.c**) return TRUE iff name specifies a directory.

f_iswild() (**f_util.c**) return TRUE iff name contains DOS wildcard characters.

f_joinname() (**f_util.c**) join directory path name and file name to create a full path name.

f_loadcfg() (**f_util.c**) Load the PCL configuration file into the form provided; obviously we don't require that all entries in the file be in the form. Return TRUE if file loaded.

f_matches() (**f_util.c**) return TRUE if wildcarded name pattern matches the given name. This is supposed to be DOS equivalent, whatever that is (it seems to be rather ill-defined and inconsistent, but then, this is DOS we are talking about here).

f_mlwrite() (**forms.c**) MultiLineWrite: Write a set of '\n' separated strings to the screen - start a new row after each '\n';

f_nelem() (**forms.c**) return the number of elements currently set in the vector of values for the variable. Return 0 if not values set.

f_path() (**f_util.c**) search ';' separated path of directories for file of "name". Stick full name in "full", and return it, if found..

f_select() (**f_util.c**) Allow the user to select a file. Prompt for name, using default name, if provided. If the user responds with a wild-carded name, then display file selection menus. When a menu is exited without a choice, reprompt. If a prompt is exited without a response, return NULL. Else return the name typed or selected.

f_selecto() (**f_util.c**) Let the user select an output file name. If the file exists, offer to append or overwrite the file.

f_splitname() (**f_util.c**) split the full path name into directory and file names. Allow for drive names on the front (they are part of directory name).

`f_valcol()` (`formmeth.c`) return the column number at the start of the value part of the field. This leaves room for the label before the value.

`f_valrow()` (`formmeth.c`) return the row number at the start of the value part of the field. This puts the value on the last line of multi-line labels.

`field_in()` (`formsio.c`) Use the text string to set the values of the given field. Use the vector flag and subscripts returned by line_in() to set all elements of vectors.

`field_out()` (`formsio.c`) write one field to the output file. Field options may state that the field is not to be written – honour that. Output file ID first on the line, then the subscript range if the field is a vector, then print all values, unless there are no values, in which case output a single "?".

`find_field()` (`forms.c`) return a pointer to a filed descriptor for the field containing the given help ID string.

`find_var()` (`forms.c`) Find the field referencing the given variable. Return a pointer to the field descriptor.

`float_check()` (`formmeth.c`) Check the float value for validity against the limits.

`float_display()` (`formmeth.c`) display an element of a float field.

`float_edit()` (`formmeth.c`) Allow the user to edit an element of a float field.

`float_erase()` (`formmeth.c`) erase an element of a float field.

`float_store()` (`formmeth.c`) store the float (or string) value for the i'th element of an float field.

`float_string()` (`formmeth.c`) return the character representation of an element of a float field.

`float_value()` (`formmeth.c`) Convert characters to value for an element of a float field.

`form_disp()` (`forms.c`) display the data values in the form, but do not allow the user to edit it.

`form_do()` (`forms.c`) display the form, and let the user enter data until she is done. This is the main form routine called by applications. Return an indication of whether form data was actually changed or not.

`form_in()` (`formsio.c`) input a form from the named file. Try to display reasonable error messages.

`form_inf()` (`formsio.c`) input a form from the opened file

`form_out()` (`formsio.c`) Output a form to the named file, in replace or append mode, depending on options.

`form_outf()` (`formsio.c`) Output the form to the opened file. If the mode is not given as "append", write out a small header to identify the type of form file.

`fpr()` (`estperf.c`) Print a floating point number on the output stream, specially marking NAN's (Not-A-Numbers).

`freedir()` (`scandir.c`) Free all the space allocated by scandir.

`g_los()` (`tables.c`) Return the level of service (name) for the level of service index, approach.

`g_lwid()` (`tables.c`) Return the lane width string for the given lane width index.

`g_nlos()` (`tables.c`) Return the number of different levels of service, approach.

`g_nlwids()` (`tables.c`) Return the number of different lane widths in the tables, approach.

`g_npnpz()` (`tables.c`) Return the number of different % no passing zones in the approach v/c table.

`g_nswids()` (`tables.c`) Return the number of different shoulder widths in the tables, approach.

`g_swid()` (`tables.c`) Return the shoulder width string for the shoulder width index.

`gdist()` (`cl_comp.c`) Interpolate between two points on the speed-distance curve and return the distance at which the speed equals "ispeed"

`getCoefs()` (`estpass.c`) Read the coefficients for all of the regression equations from the data file. Use the percent of traffic in direction one to get the "average" speed in both directions.

`get_file_choice()` (`f_util.c`) given an array of possibilites (all files and/or directories), display a menu of them to the user, and return the users selection. If user does not want to make a selection, return NULL. Show Directory names in the menu with a trailing slash. The possibilities have already been filtered and sorted by name.

`get_input()` (`defcl.c`) Read the configuration file. Then, if no site file has been specified, ask the user to specify one. Finally, read the site file to obtain the site parameters, and the results previously computed by this module (if any).

`get_input()` (`defsite.c`) Load the configuration file (to set some defaults), and any existing site file. If a site file has not been specified, default the name and ask the user to confirm and or supply a file name.

`get_input()` (`effmeas.c`) Load the configuration and site files. Default the name and prompt the user for the site file if not specified. Complain and die if lane performance data is not available.

`get_input()` (`estpass.c`) load configuration and site files. If site not specified, prompt user after defaulting site file name. Complain if lane geometry not avalibale in site file. Determine number of cases, and expand the peak-hourly-factor array, if necessary.

`get_input()` (`estperf.c`) load configuration and site files. If site not specified, prompt user after defaulting site file name. Complain if lane geometry not avalibale in site file. Determine number of cases, and expand the peak-hourly-factor array, if necessary.

`get_input()` (`pcl.c`) load the configuration file, which is simply a read-only form defined by a table.

`get_rv()` (`effmeas.c`) Calculate relative values of the given effectiveness measures. That is, normalize by dividing each value in the vector of numbers by the maximum value in the vector.

`get_time()` (`defcl.c`) Get the current time, ready for printing on output.

`get_time()` (`effmeas.c`) Get the current time, ready for printing on all output.

`get_time()` (`estpass.c`) Get the current time, ready for printing.

`get_time()` (`estperf.c`) Get the current time, ready for printing.

`gr2i()` (`cl_comp.c`) Calculate which integer grades are needed from the file in order to compute the speed-distance curves for the given grade. Fractional grades require 2 integer grades because of interpolation.

`gr2int()` (`tables.c`) Return integer grade, in %, from floating point value: take next higher grade.

`gr_drawtics()` (`graph.c`) draw and label tic marks on both axes of a graph.

`gr_findtic()` (`graph.c`) Find appropriate interval for tic marks, and adjust xmin and ymin so that an integer number of tic intervals will fir the range. The number of marks will not be greater than that specified.

`gr_graph()` (`graph.c`) Draw a graph of the supplied data. "nx" is the number of points in the array of x-values, and "x" are the points. "ny" is the number of SETS of y-values, each SET makes up one curve on the graph. "yvec" is an array of pointers to arrays of numbers, each of which contains "nx" data values.

`gr_init()` (`graph.c`) Initialize the graphics system. Determine device coordinate ranges, and set the transformations to identity.

`gr_line()` (`graph.c`) Draw a line from pt (x0,y0) to (x1,y1). (x1,y1) becomes the new pen position.

`gr_lineto()` (`graph.c`) Move the "pen" to the given position, drawing a straight line from the current position to the given one.

`gr_moveto()` (`graph.c`) Move the graphics "pen" to the given position, without drawing a line.

`gr_rect()` (`graph.c`) Draw a filled rectangle, given lower left and upper right coordinates. "pattern" specifies the fill pattern to use (currently, only a slash is permitted).

`gr_setdev()` (`graph.c`) pop up a menu so that a user can select one of the available graphics output devices. Set the current device to that selection.

`gr_setfont()` (`graph.c`) Set the current font parmeters to those given, including direction and size. Currently, we only support a sans serif font, in horizontal or vertical direction.

`gr_setlabeler()` (`graph.c`) set the point labeler function. This is a user function to call that returns a string used to label each point on each curve.

`gr_setls()` (`graph.c`) Set line style and thickness.

`gr_term()` (`graph.c`) terminate graphics system. Reset transformations, prompt user to strike a key and wait for it. Clear graphics screen and restore screen to text mode.

`gr_text()` (`graph.c`)

`gr_viewport()` (`graph.c`) Set the viewport, which is the portion of the screen to use for subsequent graphics. The lower left and upper right corner are given in normalized device coordinates (i.e., (0,0) at lower left, (1,1) at upper right). This is used to set some of the transformation parameters.

`gr_window()` (`graph.c`) Set the window dimensions, which is that portion of the users world coordinates that are to be displayed in the viewport. The lower left and upper right corners of the window are given in the users coordinate system. Transformations will be set so that these points coincide with the ll and ur corners of the viewport.

`grade_check()` (`defsite.c`) check all of the grade distances and grades for consistency and validity.

`gspeed()` (`cl_comp.c`) Interpolate between two points on the speed-distance curve and return the speed at which the distance equals "idist"

`h_ctrlfn()` (`help.c`) set the current key control function. A key control function is called (by the Blaise TOOLS) for every keystroke, before that keystroke is otherwised processed.

`h_curwin()` (`help.c`) set the current window. This is the window saved when the F10 menu "Save Window" is selected.

`h_display()` (`help.c`) display a help window corresponding to the given module and topic IDs.

`h_exitfn()` (`help.c`) set the current exit function. This is the applications function that gets called when an exit is requested (either through the F10 options menu, or through Alt-X).

`h_helpfn()` (`help.c`) set the current help function. This is the function that gets called to service F1 and Ctrl-F1 for menus. This function should display help based on the currently highlighted menu item.

`h_init()` (`help.c`) initialize the help system

`h_keyfn()` (`help.c`) define a new special function key ("hot" key), with a function to get called whenever that key is pressed.

`h_keys()` (`help.c`) set the current KEYS id. This is used to describe the type of input (menu, number, etc.), and what special keys may be pressed.

`h_menu_kcf()` (`help.c`) return a pointer to the menu key control function.

`h_menufn()` (`help.c`) set the current menu function. This is the function that gets called to service an F10 options menu request. A default function is provided.

`h_mitem()` (`help.c`) add an item to or change an existing item in the F10 options menu. Specify also the help file topic ID, and the "lotus-style" documentation string, and the function to call when the item is selected.

`h_module()` (`help.c`) set the current module id and module name.

`h_norm_kcf()` (`help.c`) return a pointer to the normal key control function (used for everything except menus)

`h_restore()` (`help.c`) restore the current state of the help system from an application-supplied buffer.

`h_save()` (`help.c`) save the current state of the help system in an application-supplied buffer.

`h_statline()` (`help.c`) display the normal status line, with str being used as the module name, also displayed.

`h_topic()` (`help.c`) set the current topic id. This is used to explain the current input request from the point of view of the application.

`h_wprint()` (`help.c`) Save the contents of the window (or of the current window) in a file. Enter into a dialogue with the user to get the name of the file. This saves the text without attributes or colours, and the results are thus suitable for printing on almost any printer.

`helpsys()` (`helpsys.c`) This is the only entry point into the introductory help system. It display a menu of choices, while holding the user by the hand.

`hf()` (`estpass.c`) Calculate the hf (headway factor) used to compute assured passing opportunities without a passing lane. Interpolate values from the table of HF values.

`hide_field()` (`forms.c`) The field marked by the given help ID is to be hidden. Hide it.

`hook_edit()` (`formmeth.c`) edit an element for a hook field. Actually, just call the applications verify function, if one exists. This allows us to create invisible form fields that perform data verification after a complete set of data has been entered. It is assumed that this field occurs AFTER the data fields to be checked.

`init_calc()` (`estpass.c`) Initialize for all computations.

`init_keys()` (`init.c`) Cahnge the default behavior of some of the special keys, such as Alt-X, Alt-U, Alt-G, Up, Down, Tab and Shift-Tab.

`init_sections()` (`estperf.c`) Initialize some global data. Calculate vehicles per hour, AADT's for the case study (different DHV's). Build an array of SECTIONS, where each section is portion of roadway of constant grade. Calculate an average grade over entire site.

`initial_help()` (`pcl.c`) Depending on help level, offer to start the initial help system.

`input_it()` (`defcl.c`) Ask the user to supply start and end distances for climbing lane.

`int_check()` (`formmeth.c`) Check the integer value for validity against the limits.

`int_display()` (`formmeth.c`) display an element of an integer field.

`int_edit()` (`formmeth.c`) edit an element of an integer field.

`int_erase()` (`formmeth.c`) erase an element of an integer field.

`int_store()` (`formmeth.c`) store the int (or string) value for the i'th element of an int field.

`int_string()` (`formmeth.c`) convert integer value to characters for an element of an integer field.

`int_value()` (`formmeth.c`) convert characters to integer value for an element of an integer field.

`interp()` (`cl_comp.c`) Create a new speed-distance curve for grade "gr" by linear interpolation between 2 existing ones for grades "gr0" and "gr1". Return a pointer to the array of points, and set the number of the points. If there are not the same number of points in the curves, extend the shorter by assuming constant speed.

`is_blank()` (`formsio.c`) return TRUE if string s is completely blank

`is_cl()` (`pcl.c`) answer TRUE iff the site is to be analysed for a climbing lane, else answer FALSE to indicate a passing lane.

`is_downkey()` (`formmeth.c`) Anser TRUE if given key is one that indicates we should move DOWN or AHEAD in the form.

`is_upkey()` (`formmeth.c`) Answer TRUE if key is one that signals we should move UP or BACK in the form.

`l2c()` (`forms.c`) Format a range descriptor into a printable form and return a pointer to the static character string.

`l_interp1()` (`tables.c`) Linearly interpolate a value from the table of values. "x" is an array of x-values in ascending (or descending) order. "y" is an array of corresponding y-values, "n" is the number in each. "xval" is the input x-value. Function returns interpolated y-value. If "xval" is outside the range of x-values in the array, options control whether the interpolation is still linear, using the last two values, or if the y-values are 'flattened' (i.e., using the y-value of the last entry).

`lgetc()` (`estpass.c`) read one line of coefficients form the data file.

`lgets()` (`estpass.c`) read one line of characters from the data file, ignoring comments, blank lines, etc.

`line_in()` (`formsio.c`) Read one line from input file, discarding blank and comment lines. Extract file ID of file, and subscript ranges of vectors, if appropriate. Copy ID and value portion to callers buffers. Return 0 if OK.

`load_file()` (`cl_comp.c`) Load the required columns (grades) from the file whose base names is "base" and whose extension is "ext". The data file is assumed to contain data for integer grades from "igr_min" to "igr_max" in increments of +1%. Load and store only those columns for which the load_mask[] term is non-zero. Store the data in newly allocated arrays, which are pointed to by the "in_data" array.

`log_open()` (`defcl.c`) Open a log file so that we can start logging intermediate results. Ask the user to supply the name of the log file. Default the name from the site ID. Open the file to append data (write a form feed if the file existed previously).

`log_open()` (`effmeas.c`) Open a log file and start logging the intermediate results. Ask the user to supply the log output file name (with a suitable default). If log file alread exists, confirm the choice then append a form feed and the new results.

`log_open()` (`estpass.c`) Ask user for a file name, open a log file to append data.

`log_open()` (`estperf.c`) Ask user for a file name, open a log file to append data.

`log_selector()` (`defcl.c`) Change the pane menu to have a selector allowing one to start (or stop) intermediate result logging.

`log_selector()` (`effmeas.c`) Add an intermediate result logging item to the pane menu. The wording should reflect the current state of logging.

`log_selector()` (`estpass.c`) Add a logging selector item to the pane menu. The wording reflects the current state of the toggle switch.

`log_selector()` (`estperf.c`) Add a logging selector item to the pane menu. The wording reflects the current state of the toggle switch.

`log_toggle()` (`defcl.c`) Toggle the state of intermediate result logging.

`log_toggle()` (`effmeas.c`) Toggle the state of logging. Change the pane menu selector to reflect the change.

`log_toggle()` (`estpass.c`) Change the state of the logging toggle switch. Change to pane menu selector item to reflect the change.

`log_toggle()` (`estperf.c`) Change the state of the logging toggle switch. Change to pane menu selector item to reflect the change.

`los_as()` (`estpass.c`) return level of service for a given average speed.

`los_pp()` (`estpass.c`) return level of service for a given percent platooning.

`lwrite()` (`help.c`) Write a line to the file. Change non-printable chars to blank, trim trailing blanks, insert new-line, trim trailing blank lines.

`m_getspot()` (`m_util.c`) Return the row and column number of the currently highlighted menu item.

`m_moff()` (`m_util.c`) Turn off mouse handling (Currently not implemented).

`m_mon()` (`m_util.c`) Turn on mouse handling. (Currently not implemented).

`m_protect()` (`m_util.c`) For the given menu, specify that the selector whose ID is the same as "id" is protected (not selectable).

`m_vchoose()` (`m_util.c`) Popup the menu described by "item" and return the number corresponding to the users choice. Make the menu vertical, and Lotus-style (i.e., display doc strings) (unless asked not to).

`main_loop()` (`pcl.c`) Initialize the screen and windows, initialize the help system, offer to display the initial help system, then display the main menu and interact with the user.

`make_wild()` (`f_util.c`) make a file name into a wildcard name by replacing the name component with "*" and leaving the extension.

`maxlen()` (`formmeth.c`) Return the maximum row length of '\n' separated strings.

`menu_kcf()` (`help.c`) Menu key control function. Key control function for use when a menu is active. This function calls a menu-dependant function to service F1 and Ctrl-F1; that function should determine which item is currently highlighted, and display help based on that. If neither F1 nor Ctrl-F1 is detected, norm_kcf() is called to handle the other special keys.

`norm_kcf()` (`help.c`) normal key control function. Key control function to use in most contexts. This function checks for many special function keys, and handles some specific ones such as Alt-H, F1, Ctrl-F1, Alt-F1, F10, etc. It can also check for a small number of application defined special keys, and will call an application supplied function for each of those.

`open_file()` (`cl_comp.c`) Open a speed-distance table file by looking in several places, then check the file to see that it is a speed-distance file. Return the file pointer and its type (text or binary)

`pass_fn()` (`defsite.c`) This function is called when the user selects the "passing allowed" field. The form is adjusted based on whether passing is allowed or not (if it is allowed, we must ask for percent no-passing).

`percent_check()` (`defsite.c`) Check the percent of trucks and RV's for reasonableness.

`pfvf()` (`effmeas.c`) Print a vector of floating point numbers as decimal numbers with "n" decimals each. Print all numbers across on one line. Print head and tail strings before and after.

`pfvi()` (`effmeas.c`) Print a floating point vector as a vector of integers across one line. Print head and tail character strings before and after the numbers.

`plot_air()` (`effmeas.c`) Plot accident reduction as a function of cost/100 veh.

`plot_dr()` (`effmeas.c`) Plot delay reductions as a function of $/100 veh.

`plot_eff()` (`effmeas.c`) Plot weighted effectiveness as a function of cost/100 veh.

`plot_net()` (`effmeas.c`) Plot net benefits as a function of dhv.

`plot_results()` (`defcl.c`) Plot the computed results on the screen. Save state of CRT before plotting, then restore it after.

`prev_choice()` (`m_util.c`) Return the choice made the last time this menu was displayed, or 0 if never displayed before.

`print_PPAS()` (`estpass.c`) calculate and print percent platooning and average speeds, without and with passing lane.

`print_header()` (`effmeas.c`) Print some header data on the intermediate result log.

`print_header()` (`estpass.c`) Print header on the log output file.

`print_header()` (`estperf.c`) Print header on the log output file.

`print_intro()` (`defcl.c`) Print the header and site info on the screen.

`print_log()` (`defcl.c`) Print header and site info on log file.

`print_results()` (`defcl.c`) Print the computed results on the screen.

`protect()` (`m_util.c`) For a given menu selection, see if it is protected or not. Only if the current menu is the same as the one for which the protection list is defined.

`pv2str()` (`util.c`) Convert vector of pointers to strings ("pv") to blank separated string in "buf".

`quit()` (`defcl.c`) Terminate execution. If data has been computed or changed, offer to save it first. Restore screen.

`quit()` (`defsite.c`) Terminate execution. If data has been changed, offer to save it. Restore the screen.

`quit()` (`effmeas.c`) Terminate execution. If results have been computed, offer to save them to a print file. Write the communications file. Restore the CRT screen.

`quit()` (`estpass.c`) Terminate execution. If data has been computed, offer to save it. Write com file and restore CRT.

`quit()` (`estperf.c`) Terminate execution. If data has been computed, offer to save it. Write com file and restore CRT.

`read_com()` (`pcl.c`) read the communication file written by another module. That file contains site ID, site description, and site file name.

`save_data()` (`defcl.c`) Save the results computed by this module. If there is more than one zone of climbing lane, ask user if she wishes to use only the first zone. If there are no zones, ask the user about that, too. If no site file has been specified, default it from site ID. Ask user to confirm that data will be appended to site file. Then append the data to the site file.

`save_data()` (`defsite.c`) Save the entered data into the site file. Default the name of the file from the site ID, then confirm the name with the user.

`save_data()` (`effmeas.c`) Copy all text in the virtual window to a file of the users choosing.

`save_data()` (`estpass.c`) Save user input and computed performance data into site file. If site file not specified, default from site ID. Confirm that user wishes to append this data to site file.

`save_data()` (`estperf.c`) Save user input and computed performance data into site file. If site file not specified, default from site ID. Confirm that user wishes to append this data to site file.

`save_fn()` (`defsite.c`) This function is called when the user selects the "Save ?" field. If saving is indicated, the form is written out to the site file.

`scandir()` (`scandir.c`) Read the directory whose name is "dirname", and build a sorted list of direct entries, each entry describing a file or a sub-directory. Return the number of entries found. If "select" is non-null, it is a pointer to a function that is called for each entry, used to filter out unwanted entries. "compare" is a function to call to compare names (or whatever) for sorting; it defaults to an alphabetic sort by name.

`sd_load()` (`cl_comp.c`) Initialize this module and load all of the the speed-distance curves needed for the site.

`show_LF()` (`estpass.c`) display a table of suggested effective distances for various queue sizes and traffic volumes. This information is useful to the user when filling out the effective distance part of the input form.

`show_field()` (`forms.c`) The field containing the given help ID is not to be hidden any longer. Show it.

`show_limits()` (`forms.c`) Show the value limits for the field currently being editted.

`sl_beep()` (`statline.c`) Sound an alarm.

`sl_clear()` (`statline.c`) Clear the status line to blanks.

`sl_err()` (`statline.c`) Display a formatted error message in the status line, beep, wait for a keystroke, then restore the previous contents of the status line. Use "module" and "topic" as ID's for the help system, if the user requests help before typing a character to clear the status line. The help topic will presumably explain the error message in more detail.

`sl_inited()` (`statline.c`) return TRUE iff status line has been initialized.

`sl_msg()` (`statline.c`) Display the formatted message in the status line. The format and arguments are like those of "printf()".

sl_restore() (**statline.c**) Restore the satus line contents from the callers buffer.

sl_save() (**statline.c**) Save the current status line contents into the callers buffer.

sl_set_all() (**statline.c**) Set remainder of status line, from "col" to end, to be character "ch" with given foregorund and background colours.

sl_setrow() (**statline.c**) Set the status line row # on screen to be that given.

str2float() (**util.c**) convert string in "buf" to a floating point value, store in "*varp" only if O.K. Return 0 if O.K., error column+1 if error.

str2long() (**util.c**) convert string in "buf" to a long integer value, store in "*varp" only if O.K. Return 0 if O.K., error column+1 if error.

str2pv() (**util.c**) Convert the space separated strings in "str" to a pointer vector (array of pointers to strings). Use "buf" to place the null-terminated strings in. Place a NULL pointer after the last one in "pv".

str_check() (**formmeth.c**) Check the string value for validity.

str_display() (**formmeth.c**) display the i'th element of a string field. Return the number of characters displayed.

str_edit() (**formmeth.c**) Edit the i'th element of a string field.

str_erase() (**formmeth.c**) Erase the i'th element of a string field.

str_store() (**formmeth.c**) store the string value for the i'th element of a string field.

str_string() (**formmeth.c**) Return the character representation of the i'th element of a string field.

str_value() (**formmeth.c**) Convert the character representation of the i'th element of a string field to characters.

strhgt() (**formmeth.c**) Return the number of rows that will be printed from a '\n' separated string (essentially, number of '\n' + 1).

`summarize_sd()` (`cl_comp.c`) Summarize the speed-distance curves for all sections, on the log file

`summarize_tables()` (`cl_comp.c`) Print a summary of the speed-distance tables on the log file

`summarize_zones()` (`cl_comp.c`) Print a summary of all zones found, on the log file.

`t_time()` (`cl_comp.c`) Compute the travel time, in hours, between 2 points on the speed-distance curve. Assume constant acceleration between the two points, so just use average speed.

`terr_check()` (`defsite.c`) This function is called after the user answers the "terrain type" field. The form is adjusted based on whether the terrain is hilly or not (if it is hilly, we must ask for percent grades).

`title_check()` (`formmeth.c`) Check the title value for validity (NO-OP).

`title_display()` (`formmeth.c`) display an element for a title field (NO-OP)

`title_edit()` (`formmeth.c`) edit an element for a title field. (NO-OP)

`title_erase()` (`formmeth.c`) erase an element for a title field (NO-OP)

`title_store()` (`formmeth.c`) store the string value for the i'th element of a title field (NO-OP).

`title_string()` (`formmeth.c`) get characters for an element for a title field (NO-OP)

`title_value()` (`formmeth.c`) convert characters for an element for a title field to an internal value (NO-OP)

`trim_curve()` (`cl_comp.c`) Trim one array of speed-distance points by looking for the end of the curve, (i.e, the position at which the speed ceases to change). This provides the maximum sustainable speed on the grade. Update the number of pts to be that of the first occurrence of that constant speed. Be careful to leave at least two points in the curve.

`trim_data()` (`cl_comp.c`) Trim all loaded speed-distance curves (i.e., find the flat portion of each).

`u_aus()` (`tables.c`) Return the average upgrade speed for the given grade and speed index (upgrade).

`u_banner()` (`util.c`) print a banner identifying the name, version and release date of the program.

`u_fd()` (`tables.c`) Return the "fd" factor (directional distribution adjustment) for the given % upgrade.

`u_fw()` (`tables.c`) Return the "fw" factor (lane and shoulder width adjustment) for the given level of service, lane width and shoulder width indices.

`u_gsi2row()` (`tables.c`) Return the row number in the table corresponding to the given grade and speed index (upgrade).

`u_ilos()` (`tables.c`) Return the level of service index corresponding to the given average speed, upgrade.

`u_ilos2()` (`tables.c`) Return the level of service index for the given grade and average upgrade speed index (upgrade).

`u_los()` (`tables.c`) Return the level of service name for the given level of service index, upgrade.

`u_naus()` (`tables.c`) Return the number of different average speeds in the v/c table for the given grade (i.e., number of rows for the grade) (upgrade).

`u_pce()` (`tables.c`) Return the "E" (or "E0") value (passenger-car equivalents for trucks on grade) for the given grade, length of grade, and average upgrade speed.

`u_progid()` (`util.c`) Return a string containing name and version of PCL, and module name (if supplied).

`u_vcr()` (`tables.c`) Return the "v/c" ratio for the given % no passing, grade, and speed index (upgrade).

`ugs()` (`estperf.c`) Adjust the average speed upgrade with climbing lane. The regression equations may give values that are outside the boundaries given by other parmeters.

`usage()` (`defcl.c`) Print usage message and die.

`usage()` (`defsite.c`) Print a usage message and die.

`usage()` (`effmeas.c`) Print a usage message and die.

`usage()` (`estpass.c`) Print a usage message and die.

`usage()` (`estperf.c`) Print a usage message and die.

`use_avg()` (`defsite.c`) An average grade is to be used instead of the grades input by the user. Compute that average and update the dist and grade arrays for the new situation. This function is called whenever the user responds to the "Use average grade" question.

`value_erase()` (`formmeth.c`) erase the i'th element of the field.

`vchoose_helper()` (`m_util.c`) Display a help screen based on the currently selected menu item. This function is given to the help system as the help function before a menu is read.

`vdisplay_fields()` (`forms.c`) display the label and all elements of a field (unless the field is hidden).

`vedit_fields()` (`forms.c`) Edit all the elements of a field.

`verase_fields()` (`forms.c`) erase the label and all elements of a field.

`vw_make()` (`w_util.c`) Create and return a new virtula windo of given size, title, options, and data area size. A virtual window can contain a data area that is bigger than the viewport; thus it is a scrollable window.

`w_air_check()` (`effmeas.c`) Check the weights for delay reduction and accident involvment reduction measures.

`w_air_default()` (`effmeas.c`) Compute the default weight for the accident involvement reduction measure.

`w_confirm()` (`w_util.c`) Popup a 'confirmer' for the user to respond to. "topic" is the help file topic ID. "prompt" is the question.

`w_confirmat()` (`w_util.c`) Pop up a confirmer at a given screen location.

`w_make()` (`w_util.c`) Create and return a new window of given size, title, and options.

`w_prompt()` (`w_util.c`) Use a prompter to ask user for arbitrary text. "keys" will be used as the Help file 'KEYS' topic ID.

`w_restore()` (`w_util.c`) Make the window whose state was earlier saved in the callers buffer the currently active window (got that?).

`w_save()` (`w_util.c`) Save state about the currently active window into the callers buffer.

`w_setlocn()` (`w_util.c`) Within existing window "win", a new window of size "height" by "width" is to be created. Set the location values in "locn" appropriately, depending on the options given in "opts". We like to place the new window near the current cursor location, but other options and domensions may not allow that.

`w_yesno()` (`w_util.c`) Popup a small menu from which the user can select 'yes' or 'no'.

`wp_file()` (`w_util.c`) Prompt user for a file name.

`wp_float()` (`w_util.c`) Prompt user for a floating point number.

`wp_text()` (`w_util.c`) Prompt user for arbitrary text, supplying default value.

`write_com()` (`defcl.c`) Write the communication file so that we can tell the invoking program what we worked on. Write site ID, description, and site file name into the comm file.

`write_com()` (`defsite.c`) Write site ID, description and file name into com file in order to tell invoking program what we did.

`write_com()` (`effmeas.c`) Write the communication file to tell the invoking program what we did.

`write_com()` (`estpass.c`) Write info to communication file to tell invoking program what we worked on.

`write_com()` (`estperf.c`) Write info to communication file to tell invoking program what we worked on.

`xfree()` (`util.c`) Free memory previously allocated with xmalloc() or xrealloc().

`xgets()` (`pcl.c`) read one line from file, trimming the trailing newline.

`xmalloc()` (`util.c`) Allocate "num" contiguous units, each of size "len",.

`xrealloc()` (`util.c`) Reallocate old space to "num" contiguous units, each of size "len".

Index

T

U

W

X

Z